DSM-PM2 : une plate-forme portable pour l'implémentation de protocoles de cohérence multithreads pour systèmes à mémoire virtuellement partagée

Gabriel Antoniu

N^o d'ordre : 200
N^o bibliothèque : 01ENSL0200

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'école doctorale de MathIF

présentée et soutenue publiquement le 21 novembre 2001

par Monsieur Gabriel ANTONIU

DSM-PM2 : une plate-forme portable pour l'implémentation de protocoles de cohérence multithreads pour systèmes à mémoire virtuellement partagée

Directeurs de thèse :	Monsieur	Luc	BOUGÉ
	Monsieur	Raymond	NAMYST
Après avis de :	Monsieur	Thierry	PRIOL
	Monsieur	Pierre	SENS

Devant la commission d'examen formée de :

Monsieur	Luc	BOUGÉ, Membre
Monsieur	Denis	CAROMEL, Membre
Monsieur	Raymond	NAMYST, Membre
Monsieur	Thierry	PRIOL, Membre et Rapporteur
Monsieur	Pierre	SENS, Membre et Rapporteur

Monsieur Jean-Bernard STEFANI, Membre

À la mémoire de mes parents,

Remerciements

Ce manuscrit est le résultat de trois années de travail. Pendant ces trois années j'ai découvert un domaine de recherche intéressant que j'ai exploré et dans lequel j'ai proposé un petit incrément. C'est l'évaluation de cet incrément qui fait réunir un jury autour d'une table aujourd'hui, le 21 novembre 2001. Je remercie Thierry Priol, Pierre Sens, Denis Caromel et Jean-Bernard Stefani d'être venus des quatre coins de la France pour juger mon travail. Les remarques particulièrement constructives de Thierry et de Pierre, en leur qualité de rapporteurs, m'ont permis d'acquérir une vision légèrement différente de mon travail et de le juger avec plus de recul. Mais, tout naturellement, c'est à mes excellents directeurs de thèse, Luc Bougé et Raymond Namyst que s'adressent mes premiers remerciements.

C'est le cours de parallélisme de données de Luc, suivi en DEA, qui a tout déclenché. Savoir présenter un domaine d'une manière aussi claire et aussi attractive est un art, et Luc en est le maître. Ce cours, avec les exposés et les débats auxquels il nous faisait participer, m'a donné un aperçu de ce que peut être la vie de chercheur et m'a clairement donné envie de la choisir. Même si ce choix a été préparé par certaines expériences antérieures, ce cours a confirmé mon intuition et a mis fin a quelques hésitations. J'ai donc décidé de faire mon stage de DEA sous la direction de Luc et... c'est là que j'ai découvert Raymond, co-encadrant du stage. Les deux m'ont démontré d'une manière particulièrement convaincante comment on peut, en quelques mois, apprendre un domaine, choisir un problème et même le résoudre d'une manière qu'on n'aurait même pas imaginé au début ! J'ai donc décidé de continuer cette expérience avec eux pendant trois ans. Raymond, dont j'ai été le premier doctorant, m'a fait découvrir tout un domaine (que j'ai aimé même si ce n'était pas tout a fait celui que j'avais choisi initialement !). Avec compétence et patience, il m'a souvent montré que là où il semblait ne pas y avoir de solution, il pouvait tout de même y en avoir une. Merci Raymond, pour ton optimisme irrémédiable, pour ta jovialité et pour ta confiance constante ! De son côté, Luc m'a aidé à mieux définir ma démarche (et à mieux l'expliquer !), mais il m'a surtout fait comprendre qu'il était toujours disponible et que mes requêtes d'interaction étaient plus importantes que la plupart de ses autres tâches. Son énorme disponibilité pour toute discussion, qu'elle ait été technique ou stratégique, sur le contenu ou sur la forme, a compté pour beaucoup dans la progression de mon travail de thèse. Je le remercie très chaleureusement. Merci aussi, Raymond et Luc, pour m'avoir mis précisément dans ces situations particulièrement fertiles, comme la participation à IPDPS 1999 et les visites chez Phil Hatcher, qui ont été déterminantes pour l'orientation de ma thèse.

En effet, cette thèse est également le résultat d'autres interactions, plus ou moins choisies. Les séjours dans l'équipe de Phil Hatcher ont été particulièrement fructueuses (quatre fois coauteurs !). Merci, Phil, j'ai beaucoup apprécié ton efficacité, mais aussi ta gentillesse ! Par ailleurs, la rencontre avec Frank Muller à IPDPS 1999, suivie de mon séjour à Berlin, a été à l'origine des premières idées développées dans cette thèse. Je remercie Frank pour m'avoir tout dit sur DSM-Threads (mais aussi pour m'avoir hébergé chez lui pendant une agréable semaine et m'avoir fait découvrir Berlin et la Kristalweizen !).

Mais je dois également remercier le LIP en son ensemble, qui fournit un environnement si favorable à la recherche et si convivial ! Tout particulièrement mes remerciements vont vers mes collègues de travail. Et en premier vers Christian Pérez, mon "aîné" dans l'équipe PM2, premier

collègue de bureau et premier collaborateur. (Nos discussions joviales ont souvent fait que la journée démarre bien !) Je remercie Ioan pour m'avoir fait découvrir le LIP et pour m'avoir parlé des merveilleuses choses qui s'y passaient. Mais je remercie aussi les autres : Guillaume Huard, Olivier Reymann, Olivier Aumage (que j'aurais tant aimé rendre plus bavard sur Madeleine et autres !), Vincent Danjean (merci pour tes excellents gâteaux !), Arnaud Legrand, Frédéric Suter, Martin Quinson et mille pardons à ceux que j'oublie ! Je remercie aussi Anne-Pascale, Sylvie, Marie et Corinne pour leur assistance efficace dans les diverses démarches administratives (mais aussi pour leur bonne humeur !)

Même si l'environnement du LIP m'a été très favorable, le résultat de mon travail de thèse n'aurait peut-être pas eu été le même sans le soutien et les encouragements constants de mon épouse. Je te remercie pour tout, Magda. Les détails sont inutiles. Je remercie également tes parents pour avoir assumé, loin, des responsabilités qui me revenaient, me laissant ainsi libre de me concentrer sur mon travail.

Enfin, je remercie ceux qui ont eu une influence importante sur mes choix essentiels. Paradoxalement pour certains, peut-être, il s'agit des professeurs de littérature de mon adolescence : Elisabeta Nicolescu et Anton Chevorchian. Ma décision de faire une thèse prend ses racines dans le système de valeurs qu'ils m'ont transmis. Si, si, même si "ce qui existe existait déjà à l'état potentiel", savoir rendre réel n'est pas rien...

Résumé : Dans leur présentation traditionnelle, les systèmes à mémoire distribuée virtuellement partagée (MVP, en anglais DSM) permettent à des processus de partager un espace d'adressage commun selon un modèle de cohérence fixé : cohérence séquentielle, à la libération, etc. Les processus peuvent habituellement être distribués sur des noeuds physiquement distincts et leurs interactions par la mémoire commune sont implémentées (de manière transparente) par la MVP, en utilisant une bibliothèque de communication. Dans la plupart de travaux dans ce domaine, il est sous-entendu que la MVP et l'architecture sous-jacente sont données. Le programmeur doit alors adapter son application à ce cadre fixe, afin d'obtenir une exécution efficace. Cette approche impose des limitations statiques et ne permet pas de comparer des approches alternatives.

La contribution de cette thèse consiste à proposer une plate-forme *générique* d'implémentation et d'expérimentation appelée DSM-PM2, qui permet de développer et d'optimiser *conjointement* les applications distribuées *et* le(s) protocole(s) de cohérence de la MVP sous-jacente. Cette plate-forme, implémentée entièrement au niveau logiciel, est portable sur plusieurs architectures de grappes hautes performances. Elle fournit les briques de bases nécessaires pour implémenter et évaluer une large classe de protocoles de cohérence *multithreads* dans un cadre unifié. Trois modèles de cohérence sont actuellement supportés : la cohérence séquentielle, la cohérence à la libération et la cohérence Java. Plusieurs études de performance ont été effectuées à l'aide d'applications multithreads pour l'ensemble des protocoles proposés, sur différentes plates-formes. DSM-PM^2 a été validé par son utilisation en tant que cible d'un système de compilation Java pour des grappes appelé Hyperion.

Mots-clés : Parallélisme, processus légers, threads, mémoire virtuellement partagée, DSM, PM2, iso-adresse, migration, Hyperion, compilation Java.

Abstract: In their traditional flavor, Distributed Shared Memory (DSM) libraries allow a number of separate processes to share a common address space using a *consistency protocol* according to a semantics specified by some given *consistency model*: sequential consistency, release consistency, etc. The processes may usually be physically distributed among a number of computing nodes interconnected through some communication library. Most approaches to DSM programming assume that the DSM library and the underlying architecture are fixed, and that it is up to the programmer to fit his program with them. This static view does not allow experimentations with alternative implementations.

The contribution of this thesis consists in proposing a *generic* impementation and experimentation platform called DSM-PM^2, which allows *both* the application *and* the underlying DSM consistency protocol to be *co-designed* and tuned for performance. This platform is entirely implemented in software, in user-space. It is portable across a large number of cluster architectures. It provides the basic blocks for implementing and evaluating a large number of *multithreaded* consistency protocols within a unified framework. Three consistency models are currently supported: sequential consistency, release consistency and Java consistency. Several performances studies have been carried out with multiple multithreaded applications on different clusters, in order to evaluate the proposed consistency protocols. The platform has been validated as a target for a Java compiling system for distributed architectures, called Hyperion.

Keywords: Parallelism, threads, DSM, PM2, iso-address, migration, Hyperion, Java compiling.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Les grappes de PC, mono- ou multiprocesseurs, interconnectés par des réseaux haut-débit représentent le modèle architectural dominant pour les machines parallèles d'aujourd'hui. Pendant la dernière quinzaine d'années, de nombreux efforts de recherche ont eu pour objectif de proposer des modèles et des techniques de programmation pour ce type d'architecture, le principal défi étant de leur permettre de rivaliser en efficacité avec les supercalculateurs parallèles, plus chers, car plus difficiles à construire.

Sur les grappes de PC, un paradigme de programmation de plus en plus répandu est représenté par la multiprogrammation légère (*multithreading*). Basé sur le partage d'un certain nombre de ressources (telles que la mémoire) par plusieurs flots d'exécution concurrents (*threads*) au sein d'un même processus s'exécutant sur la machine, ce modèle convient naturellement à des architectures telles que les machines multiprocesseurs à mémoire partagée. L'implémentation des tâches parallèles par des threads (dont la gestion est particulièrement efficace) plutôt que par des processus traditionnels a contribué à une amélioration significative des performance des programmes parallèles.

Ces bonnes propriétés d'efficacité ont motivé l'utilisation des threads sur des architectures distribuées également, et en particulier sur les grappes de PC, mono- ou multiprocesseurs. Un exemple d'environnement multithread distribué qui se propose de supporter le calcul parallèle haute performance sur des grappes de PC de manière à la fois portable et efficace est PM^2. Le modèle de programmation de cet environnement est basé sur les appels de procédure à distance, dont l'exécution est prise en compte par des threads. La localisation des données et des traitements est visible au programmeur, qui doit gérer explicitement les interactions entre les nœuds de la grappe. Ce modèle de programmation est donc complexe et rend compte de la nature distribuée de l'architecture.

Une approche qui facilite la programmation des architectures distribuées consiste à utiliser un système de *mémoire virtuellement partagée*. Ce système présente au programmeur l'image d'un espace d'adressage global, accessible sur tous les nœuds, et permet la communication entre les nœuds par de simples lectures et écritures dans cette mémoire commune. Le système transforme de manière transparente ces accès en messages de communication entre les nœuds, tout en cachant au programmeur les aspects liés à la localisation des données.

Cette thèse se situe à la rencontre de deux thèmes de recherche : elle s'intéresse à la programmation des grappes à l'aide du *multithreading*, grâce à un système de *mémoire virtuellement partagée*.

Chacun de ces deux thèmes a été exploré séparément par de nombreux travaux. Ils se sont concrétisés par la réalisation d'une part d'environnements multithreads, et d'autre part de systèmes à mémoire virtuellement partagée. Néanmoins, les problématiques spécifiques dans le cas où le multithreading et la mémoire virtuellement partagée sont utilisés de manière *conjointe* ont été très peu étudiées.

1.1 Objectifs de la thèse

Les progrès effectués dans les domaines des systèmes à mémoire virtuellement partagée, tels que les définitions de nouveaux modèles de cohérence de la mémoire, ou de nouvelles implémentations plus efficaces de modèles existants, ont été traditionnellement validés par la construction de nouveaux systèmes. Ces systèmes sont souvent dédiés au concept qu'ils se proposent d'illustrer et permettent assez difficilement des études comparatives "justes", car ils diffèrent souvent par leurs modèles de programmation, par leurs fonctionnalités ou par les choix d'implémentation. De plus, les architectures sous-jacentes ne sont pas forcément les mêmes et la portabilité n'est pas toujours visée, ce qui rend les comparaisons encore plus difficiles.

À l'opposé des systèmes traditionnels de ce type, notre premier objectif a été de proposer une *plate-forme* à mémoire virtuellement partagée *ouverte*, *flexible* et *configurable*, permettant à l'utilisateur de programmer ses propres modèles de cohérence à partir d'un certain nombre de mécanismes de base réutilisables. Une telle plate-forme est particulièrement utile aux concepteurs de modèles, car elle leur permet d'implémenter et de comparer plus facilement différentes solutions. De plus, les études comparatives effectuées sur une telle plate-forme sont plus significatives que les comparaisons traditionnelles des systèmes complets, grâce à l'utilisation du même ensemble de mécanismes logiciels et matériels sous-jacents. Une autre propriété importante pour ce type de plate-forme est la *portabilité* sur différents systèmes d'exploitation et interfaces de communication. Les mêmes expérimentations peuvent alors être exécutées sur plusieurs architectures différentes.

Un autre objectif important a été de pouvoir utiliser notre plate-forme comme support d'exécution dans des systèmes de compilation de langages parallèles basés sur le concept de mémoire virtuellement partagée. En fonction du modèle de programmation associé au langage et en particulier du modèle de mémoire requis, ainsi que des informations fournies par les couches hautes du système de compilation, différentes implémentations peuvent être envisagées pour gérer la cohérence des mémoires des nœuds. Ces différentes solutions peuvent être facilement mises en œuvre à l'aide des mécanismes de base déjà disponibles et comparés.

Une caractéristique importante de notre étude est qu'elle s'effectue dans un contexte *multithread*. Ce contexte nécessite la prise en compte de la concurrence des accès des différents threads à la mémoire à différents niveaux dans la conception de la plate-forme. Certains aspects liés à cette concurrence sont gérés de manière interne à la plate-forme, d'autres restent à la charge du concepteur de protocoles.

1.2 Contributions

Cette thèse est le résultat d'un travail personnel, mais également celui d'un travail en équipe dans le cadre du projet ReMaP, au Laboratoire de l'Informatique du Parallélisme, à l'École Normale Supérieure de Lyon. Luc Bougé et Raymond Namyst m'ont constamment aidé en tant que

directeurs de thèse. Une collaboration très fructueuse a été celle avec Philip Hatcher, dans le cadre d'un contrat NSF/INRIA, pour les aspects liés à Java et en particulier au système de compilation Hyperion. Cette collaboration a donné lieu a plusieurs publications communes. Les principales contributions de cette thèse sont énumérées ci-dessous. Les publications associées (énumérées plus bas) sont indiquées entre crochets.

Mémoire virtuellement partagée générique. Nous proposons la notion de *mémoire virtuellement partagée générique*, supportant l'implémentation de protocoles de cohérence multithreads. Nous avons défini un ensemble de mécanismes génériques, nécessaires dans tout système à mémoire virtuellement partagée à base de pages. Ces mécanismes constituent une couche de base au-dessus de laquelle les protocoles peuvent être implémentés. Nous avons conçu et implémenté la plate-forme DSM-PM^2, qui se propose d'illustrer la notion de *mémoire virtuellement partagée générique* [D].

Protocoles de cohérence multithreads. Afin d'étudier la spécificité du fonctionnement d'un système à mémoire virtuellement partagée en contexte multithread, nous avons conçu et implémenté sur la plate-forme DSM-PM^2 des protocoles de cohérence multithreads pour plusieurs modèles de cohérence classiques. Certaines de nos implémentations ont été dérivées de protocoles classiques (monothread), que nous avons adaptés afin de prendre en compte les aspects liés à la concurrence. D'autres ont été conçues spécifiquement pour un contexte multithread et n'ont pas d'équivalent monothread. L'objectif étant de construire une plate-forme apte à supporter des modèles et des protocoles à caractéristiques variées, nous avons choisi d'implémenter la cohérence séquentielle (modèle de cohérence forte) et la cohérence à la libération [C,N] (modèle de cohérence faible). Chacun de ces modèles a été implémenté par deux protocoles différents [D]. La mise en œuvre de ces protocoles nous a permis en même temps de raffiner la définition des mécanismes présents dans la couche générique de la plate-forme. Nous avons validé nos protocoles par des expérimentations avec différentes applications, sur plusieurs types de grappes.

Le travail d'implémentation et de validation des deux protocoles de cohérence à la libération a été effectué en collaboration avec Vincent Bernardi, dont j'ai encadré le stage de Magistère au LIP, ENS Lyon.

Implémentation de la cohérence Java. Nous avons illustré l'utilisation de la plate-forme DSM-PM^2 comme cible d'un système de compilation en intégrant cette plate-forme dans Hyperion, un système de compilation Java pour grappes de PC. Nous avons interfacé le module de gestion de la mémoire d'Hyperion sur DSM-PM^2 grâce à deux protocoles de cohérence, qui implémentent le modèle de mémoire de Java. À l'aide de ces deux protocoles nous avons étudié deux mécanismes de détection d'accès distants aux objets Java. Nous avons validé ces deux protocoles par des expérimentations avec plusieurs applications Java, sur plusieurs plates-formes.

L'ensemble de ce travail a été réalisé en collaboration avec Philip Hatcher [A,E,F,G] dans le cadre d'un contrat NSF/INRIA. Le travail présenté dans [F] a été sélectionné parmi les 5 meilleures contributions de la conférence Euro-Par 2000.

Allocation et migration iso-adresse. Nous avons conçu et implémenté un allocateur iso-adresse (`isomalloc`) destiné à garantir la cohérence des pointeurs en cas de migration des threads. Ceci a permis la mise en œuvre d'une migration *transparente* et *préemptive* [J], que nous avons

utilisée pour l'implémentation de mécanismes d'équilibrage de charge basés sur la migration de threads. Nous avons validé cette technique dans le cadre de deux compilateurs de langages à parallélisme de données : un compilateur HPF (Adaptor) et un compilateur C* (UNH-C*), en collaboration avec Christian Pérez) [B,H,I,K,O] dans le cadre de son travail de doctorat. Nous avons ensuite étendu l'allocateur iso-adresse pour permettre l'allocation dynamique de données *partagées* dans DSM-PM^2. L'approche iso-adresse a rendu possible l'utilisation de pointeurs vers des données globalement partagées (fonctionnalité rarement proposée par les systèmes à mémoire virtuellement partagée existants).

Publications

Cette thèse a conduit à des publications. Elles sont présentées ci-dessous par catégorie.

Journaux internationaux

[A] G. ANTONIU, L. BOUGÉ, P. HATCHER, M. MACBETH, K. MCGUIGAN, R. NAMYST. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279-1297, septembre 2001. Version journal de [F].

[B] G. ANTONIU, L. BOUGÉ, R. NAMYST, C. PEREZ. Compiling data-parallel programs to a distributed runtime environment with thread isomigration. *Parallel Processing Letters*, 10(2-3):201-214, juin 2000. Numéro spécial CPC 2000. Version journal de [K].

Congrès et colloques internationaux avec comité de lecture

[C] G. ANTONIU ET L. BOUGÉ. Implementing multithreaded protocols for release consistency on top of the generic DSM-PM^2 platform. Dans *Proc. International Workshop on Cluster Computing (IWCC '01), Lect. Notes in Comp. Science*, Springer-Verlag, Mangalia, Roumanie, août 2001. IEEE. À paraître.

[D] G. ANTONIU ET L. BOUGÉ. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. Dans *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01), Lect. Notes in Comp. Science, 2026*, pages 55–70, Springer-Verlag, San Francisco, avril 2001. Workshop satellite de IPDPS 2001. IEEE TCPP.

[E] G. ANTONIU ET P. HATCHER. Remote object detection in cluster-based Java. Dans *Proc. 3rd Int. Workshop on Java for Parallel and Distributed Computing (JavaPDC '01)*, page 104, San Francisco, avril 2001. Workshop satellite de IPDPS 2001. Version complète disponible sous format éléctronique.

[F] G. ANTONIU, L. BOUGÉ, P. HATCHER, M. MACBETH, K. MCGUIGAN, R. NAMYST. Compiling multithreaded Java bytecode for distributed execution. Dans *Euro-Par 2000: Parallel Processing, Lect. Notes in Comp. Science, 1900*, pages 1039–1052, Springer-Verlag, Munich, Allemagne, août 2000. Papier distingué.

[G] G. ANTONIU, L. BOUGÉ, P. HATCHER, M. MACBETH, K. MCGUIGAN, R. NAMYST. Implementing Java consistency using a generic, multithreaded DSM runtime system. Dans *Parallel and Distributed Processing. Proc. Intl Workshop on Java for Parallel and Distributed Computing, Lect. Notes in Comp. Science, 1800*, pages 560–567, Springer-Verlag, Cancun, Mexico, mai 2000. Workshop satellite de IPDPS 2000. IEEE TCPP/ACM.

[H] G. ANTONIU ET C. PEREZ, « Using preemptive thread migration to load-balance data-parallel applications ». Dans *Euro-Par '99: Parallel Processing, Lect. Notes in Comp. Science, 1685*, pages 117–124, Springer-Verlag, Toulouse, France, août 1999.

[I] G. ANTONIU, L. BOUGÉ, C. PEREZ, « Generic load balancing for HPF programs: Application to the Flame Simulation kernel ». Dans *The 3rd Annual HPF User Group Meeting (HUG '99)*, Redondo Beach, California, août 1999.

[J] G. ANTONIU, L. BOUGÉ R. NAMYST. An efficient and transparent thread migration scheme in the PM2 runtime system. Dans *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99), Lect. Notes Comp. Science, 1586*, pages 496–510, San Juan, Puerto Rico, avril 1999. Workshop satellite IPPS/SPDP 1999. IEEE TCPP et ACM SIGARCH, Springer-Verlag.

Communications invitées dans des congrès internationaux

[K] G. ANTONIU, L. BOUGÉ, R. NAMYST, C. PEREZ. Compiling data-parallel programs to a distributed runtime environment with thread isomigration. Dans *The 1999 Intl Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, Las Vegas, NV, juin 1999. Papier invité.

Congrès et colloques internationaux sans comité de lecture

[L] G. ANTONIU, L. BOUGÉ, R. NAMYST, « DSM-PM2: a multi-protocol DSM layer for the PM2 multithreaded runtime system », *dans : Proc. 2nd Workshop on Parallel Computing for Irregular Applications (WPCIA2)*, J. Chassin de Kergommeaux, B. Lecussan, J.-F. Méhaut (réd.), Toulouse, France, janvier 2000. Workshop satellite de HPCA-6.

[M] G. ANTONIU, L. BOUGÉ, R. NAMYST, C. PEREZ, « Compiling Data-parallel Programs to A Distributed Runtime Environment with Thread Isomigration », *dans : 8th Workshop on Compilers for Parallel Computers (CPC 2000)*, Aussois, France, janvier 2000. Mise à jour de [K].

Congrès et colloques nationaux avec comité de lecture

[N] G. ANTONIU, V. BERNARDI, L. BOUGÉ. Extension de la plate-forme DSM-PM2 pour le support de protocoles de cohérence relâchée multithreads. Dans *Actes des 13es Rencontres francophones du parallélisme (RenPar 13)*, pages 175-180, Paris, La Villette, Avril 2001.

[O] G. ANTONIU ET C .PEREZ. Allocation iso-adresse pour une migration préemptive de processus légers : application à un compilateur HPF. Dans *Actes des 11èmes Rencontres francophones du parallélisme (RenPar'11)*, pages 43–48, IRISA, Rennes, France, juin 1999.

1.3 Plan du manuscrit

Le chapitre 2 présente le contexte de notre travail. Nous y introduisons brièvement la notion de *parallélisme*, les différentes architectures de machines parallèles et les moyens de programmation de ces machines. Nous nous focalisons sur la programmation des grappes de PC à l'aide du multithreading et nous présentons PM^2, un environnement de programmation multithread distribué. Nous montrons ensuite que la programmation des grappes peut être facilitée grâce à la notion de *mémoire virtuellement partagée* et nous introduisons quelques concepts liés à cette notion. Nous présentons également quelques systèmes à mémoire virtuellement partagée qui ont intégré le multithreading et nous montrons que la conception de tels systèmes serait facilitée par l'existence d'une plate-forme d'implémentation et d'expérimentation de protocoles de cohérence multithreads pour mémoire virtuellement partagée. Le travail présenté dans cette thèse vise à proposer une telle plate-forme.

Cette plate-forme, que nous avons appelée DSM-PM^2, est présentée dans le chapitre 3. Nous définissons la notion de *mémoire virtuellement partagée générique* en énumérant les mécanismes génériques, indépendants des protocoles, qui constituent une couche de base au-dessus de laquelle le concepteur peut facilement implémenter différents protocoles. Nous définissons également la notion de protocole de cohérence comme étant un ensemble de routines associées à des événements génériques. Le chapitre continue avec une description de l'architecture de DSM-PM^2 et de son principe de fonctionnement. Il introduit ensuite les protocoles de cohérence multithreads intégrés dans la bibliothèque ainsi que l'interface de programmation permettant de les utiliser et d'en définir de nouveaux.

Le chapitre 4 est centré sur quelques aspects liés à l'implémentation de DSM-PM^2 et fournit une évaluation préliminaire des performances de la plate-forme. Dans un premier temps, il présente des problèmes posés par la prise en compte du multithreading dans les protocoles de cohérence. Une autre partie du chapitre est consacrée à l'implémentation de l'allocateur dynamique de DSM-PM^2, conçu selon une approche iso-adresse, qui permet de garantir la validité des

pointeurs sur les données partagées, tout en gardant un fonctionnement simple et efficace lors des transferts des pages. Le chapitre fournit également quelques mesures préliminaires qui évaluent le coût des opérations de base ainsi que les performances relatives des protocoles intégrés.

Dans le chapitre 5 nous présentons une application typique de la plate-forme DSM-PM^2 : il s'agit de son utilisation en tant que cible d'un système de compilation Java, appelé Hyperion. L'objectif principal du projet Hyperion est de permettre l'exécution transparente d'applications multithreads Java sur grappes de PC. Pour supporter l'abstraction d'une mémoire partagée uniformément accessible par tous les threads de l'application, indépendamment de leur localisation, Hyperion utilise DSM-PM^2 à travers deux protocoles de cohérence spécifiquement conçus à partir de la spécification du modèle de mémoire de Java [38]. Le chapitre fournit également les résultats d'une évaluation de ces protocoles par des expérimentations avec 5 applications à caractéristiques différentes, exécutées sur deux grappes différentes.

Enfin, une conclusion sur nos travaux et quelques perspectives pour des recherches ultérieures sont données dans le chapitre 6.

Chapitre 2

Parallélisme à grain fin et mémoire virtuellement partagée

Dans ce chapitre, nous introduisons la notion de *parallélisme*, les différents types de machines parallèles et les moyens de programmation de ces machines (section 2.1). Nous nous intéressons à une classe particulière de machines parallèles à mémoire distribuée : les grappes de PC, pour lesquelles nous présentons une technique de programmation particulière : le multithreading (section 2.2). À titre d'exemple, nous présentons l'environnement multithread distribué PM^2, qui permet de programmer ce type d'architecture. Nous montrons ensuite que la programmation des architectures distribuées peut être facilitée grâce à la notion de *mémoire virtuellement partagée* et nous introduisons quelques concepts de base liés à cette notion, dans la section 2.3. Nous présentons également quelques systèmes multithreads à mémoire virtuellement partagée (section 2.4) et nous mettons en évidence le besoin d'une plate-forme qui fournisse un cadre unifié permettant la conception, l'implémentation et l'expérimentation de protocoles de cohérence multithreads pour mémoire virtuellement partagée. La principale contribution de cette thèse a été de proposer une telle plate-forme.

2.1 Vers les hautes performances : la voie du parallélisme

Cette section explique la notion de parallélisme et les besoins auxquels elle répond. Elle continue par une brève revue de quelques architectures parallèles représentatives. Ensuite, elle présente les approches typiques utilisées (à bas-niveau) pour programmer ces machines.

2.1.1 Besoins en puissance de calcul

Depuis l'apparition de l'ordinateur à la fin des années 1940, la puissance des processeurs n'a cessé d'augmenter. Elle double tous les 18 mois. Les avancées de la dernière décennie dans le domaine des technologies VLSI, qui ont permis l'augmentation des fréquences d'horloge, ont contribué à perpétuer ce processus. Cette régularité dans le temps est connue sous le nom de loi empirique de Moore. Pourtant, malgré cette évolution exponentielle, les utilisateurs demandent toujours plus et plus de puissance.

Dans les années 1990, le TéraFlop/s (10^{12} instructions flottantes par seconde) était visé. Avant même que cette puissance soit atteinte, le défi du PétaFlop/s (10^{15} instructions flottantes par seconde) a été lancé. Cette puissance est réclamée — entre autres — par les physiciens pour commencer à travailler sur des configurations réalistes. Ces besoins sont notamment connus à travers les applications du grand défi (*Grand Challenge)* [95, 96].

Dans la recherche de hautes performances, une solution qui permet de gagner quelques ordres de grandeur par rapport à la puissance de calcul disponible avec un processeur est le parallélisme. Le parallélisme est la collaboration de plusieurs processeurs à la résolution d'un même problème. Une définition qui exprime cette idée a été donnée en 1989 par Almasi et Gottlieb :

> Un calculateur parallèle est "une collection d'éléments de traitements qui communiquent et coopèrent pour résoudre rapidement des problèmes de grande taille".

La puissance des différents processeurs peut être cumulée : avec N processeurs (identiques), on peut espérer avoir N fois plus de puissance qu'avec un seul processeur. Cette relation est a priori indépendante de la puissance des processeurs. Ainsi, la connaissance acquise dans la mise en œuvre et l'exploitation du parallélisme n'est en général pas remise en cause avec l'évolution des processeurs. L'expérience a montré que le facteur déterminant dans l'exploitation du parallélisme est le rapport entre la puissance de calcul et les possibilités de communication. Le grain de calcul doit être adapté au coût des communications. Les solutions les plus efficaces sont généralement différentes selon le rapport entre le coût des communications et celui des calculs.

2.1.2 Architectures des machines parallèles

L'idée du parallélisme date du début de l'informatique. Flynn [30] a établi une classification des architectures des ordinateurs. Cette classification, encore en vigueur, permet un regroupement des machines en grandes familles. Il existe deux classes principales de machines parallèles.

SIMD (*Single Instruction Multiple Data*). Dans les machines de cette classe, un flot d'instructions unique est diffusé à un ensemble de processeurs synchrones. Les processeurs exécutent les mêmes instructions en même temps, sur des données différentes. Ce modèle, très en vogue vers la fin des années 1980 et le début des années 1990, est tombé en désuétude pour les machines parallèles. Cependant, il se retrouve actuellement au niveau des processeurs. C'est le modèle qui se cache derrière les architectures MMX/KNI (Intel) [46], 3DNow (AMD) [2] et AltiVec (Motorola) [73].

MIMD (*Multiple Instruction Multiple Data*). Dans le cas des machines de cette classe, chaque processeur possède un flot d'instructions indépendant. Aucune contrainte de synchronisation n'est plus imposée lors des exécutions parallèles. Deux types d'architectures de machines de cette catégorie sont classiquement distingués suivant que les processeurs ont ou non directement accès à toute la mémoire. Ces modèles sont illustrés par la figure 2.1.

Les machines multiprocesseurs à mémoire partagée (*Symmetric MultiProcessor — SMP*). Les premières machines de se type ont été construites dans les années 1960. Leur principale caractéristique est le fait que les processeurs peuvent accéder directement à toute la mémoire. Si les processeurs accèdent à toute la mémoire en temps constant, la machine est dite UMA (*Uniform Memory Access*). Sinon, elle est dite NUMA (*Non Uniform Memory Access*). Les machines multiprocesseurs à mémoire partagée ont généralement un handicap lié à leur architecture : la mémoire commune est souvent accédée via un

bus unique, qui peut devenir un sérieux goulot d'étranglement et augmenter fortement le coût des accès mémoire. Cet effet est d'autant plus gênant que le nombre de processeurs connectés au bus est grand. La conséquence en est que ce type d'architecture ne pourra être efficace qu'avec un faible nombre de processeurs, ce qui impose une sévère limitation au degré de parallélisme réellement exploitable par la machine. Afin d'éviter le goulot d'étranglement de la mémoire, des caches importants sont ajoutés. Dans les premières réalisations de ce type de machines, le matériel n'assurait pas la cohérence des caches. Depuis, les machines avec cohérence de cache matériel sont apparues. Le préfixe cc (*cache coherent*) est utilisé pour les reconnaître (SMP cc-NUMA).

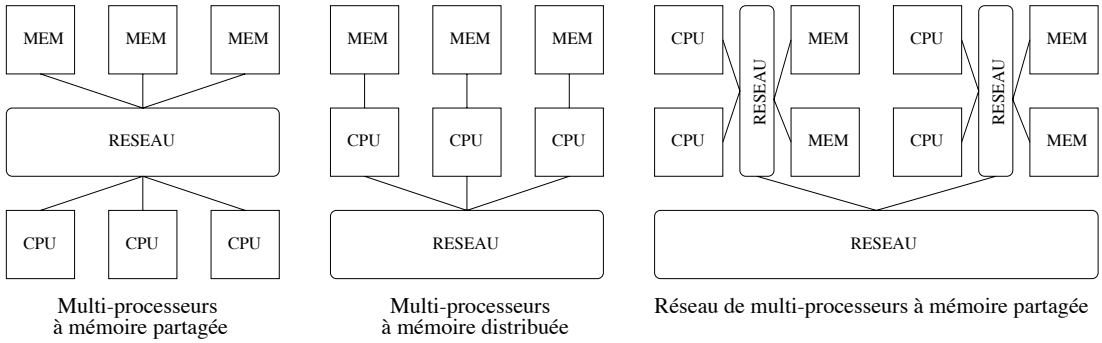


FIG. 2.1 – Les différents modèles de machines parallèles asynchrones. Les multiprocesseurs à mémoire *partagée* (SMP : Symmetric Multiprocessors) utilisent un réseau d'interconnexion (généralement un bus) entre les processeurs et la mémoire. Le temps d'accès d'un processeur aux différents éléments de mémoire peut être uniforme (UMA) ou non (NUMA). Les multiprocesseurs à mémoire *distribuée* sont un ensemble de couples processeur / mémoire reliés par un réseau d'interconnexion. Les grappes de PC implémentent ce modèle, dans lequel les communications se font par messages. Enfin, un modèle qui émerge est représenté par un mélange des deux modèles précédents : ce modèle peut être vu soit comme un réseau de machines à mémoire partagée, soit comme une machine à mémoire distribuée avec des nœuds à plusieurs processeurs. Les grappes de SMP implémentent aujourd'hui ce modèle.

Les machines à mémoire distribuée (*Distributed Memory Machines*). Dans ce type d'architecture, chaque processeur n'a accès qu'à sa mémoire locale. Les processeurs communiquent entre eux par messages via un réseau pour accéder à la mémoire des autres processeurs. L'évolution de ce type de machines s'est concentrée sur l'architecture et les performances brutes du réseau. Suivant les évolutions technologiques, les machines se sont vues dotées d'un processeur spécialisé pour la gestion des communications. L'objectif était de ne pas surcharger le processeur principal avec le coût du traitement de routage et de réception des messages. Les communications sont réalisées au niveau bas par passage de messages. Dans la plupart des implémentations, ce mode de communication est directement utilisable par le programmeur.

Jusqu'à récemment, les machines MIMD étaient de manière non ambiguë soit à mémoire partagée soit à mémoire distribuée. Actuellement, nous assistons à un mélange de ces deux modèles. Ce nouveau modèle peut être vu soit comme un réseau de machines à mémoire partagée, soit comme une machine à mémoire distribuée avec des nœuds à plusieurs pro-

cesseurs. De nombreux efforts de recherche sont en cours pour déterminer les méthode de programmation les plus efficaces pour ce type de machines.

2.1.3 Programmation des machines parallèles

Suivant les évolutions technologiques, les machines les plus performantes ont été tantôt des machines à mémoire partagée, tantôt des machines à mémoire distribuée. La programmation directe de ces machines étant différente, l'effort du programmeur devient important, car cette situation nécessite le maintien de plusieurs versions d'un même code. En plus de la version séquentielle, il faut une version spécialisée pour chacun de ces deux types de machines. En effet, le modèle de programmation des machines à mémoire partagée est la multiprogrammation. Les machines à mémoire distribuée se programment classiquement en utilisant des bibliothèques de passage de messages.

La *multiprogrammation* est basée sur l'exécution de plusieurs flots d'instructions partageant une zone de données commune. Une première implémentation a été réalisée à l'aide des segments de mémoire partagée entre plusieurs processus [91]. Actuellement, les processus légers (ou threads) représentent le moyen le plus performant de multiprogrammation [61]. L'interface de programmation des processus légers a été standardisée pour les opérations de bases (Norme POSIX 1003 [43]). Ce modèle de programmation implique l'utilisation de verrous pour l'accès cohérent aux données et nécessite également d'autres mécanismes de synchronisation plus complexes. Pourtant, ce modèle de programmation reste relativement facile à maîtriser, car il représente une extension naturelle du modèle de programmation des machines monoprocesseurs traditionnelles.

Le *passage de messages* est, comme son nom l'indique, l'envoi et la réception explicites de messages. Les processeurs, n'ayant pas accès à la mémoire des autres, ils doivent s'échanger des données via des messages. De très nombreuses bibliothèques de messages existent. L'effort de standardisation fut commencé avec l'apparition de PVM [34]. Les deux standards actuels sont PVM et surtout MPI [97]. Ce modèle de programmation est donc plus complexe, car il fait apparaître au programmeur la nature distribuée de l'architecture sous-jacente et donc, à travers cela, des problèmes liés à la localisation et à la distribution des données. La gestion de ces aspects revient alors à la charge du programmeur, alors que ce problème ne se pose pas dans le cas des machines à mémoire partagée.

La multiprogrammation et le passage de messages sont des méthodes de programmation proches du matériel. Ils permettent de très bien exploiter une architecture donnée. Cependant, comme ils sont proches du matériel, ils ne sont pas portables d'un modèle à un autre. Même à l'intérieur d'un modèle, la portabilité des performances n'est pas assurée car les programmeurs ont tendance à utiliser les spécificités de la machines dont ils disposent. Ainsi, ils augmentent les performances de leur application sur une machine au prix de la portabilité.

2.1.4 Portabilité et mémoire virtuellement partagée

Une solution à ce problème de portabilité est apportée par les *langages parallèles*. Ces langages, dont une catégorie importante est représentée par les langages à parallélisme de données, tels que HPF, C*, HyperC, etc., proposent des abstractions de haut niveau permettant la programmation simplifiée d'une machine parallèle virtuelle. Les détails liés à l'architecture physique tels que le

nombre de processeurs disponibles où la localisation des traitements sont cachés au programmeur. La responsabilité de l'implémentation des manipulations de haut niveau spécifiées par le programmeur en instructions spécifiques à l'architecture sous-jacente revient à des systèmes de compilation complexes, qui intègrent des exécutifs efficaces pour la gestion d'un certain nombre d'opérations critiques, telles que les communications ou les synchronisations.

L'approche que nous venons de décrire présente un inconvénient : elle accroît la portabilité au prix de l'acceptation par le programmeur d'un nouveau langage et de nouveaux concepts de programmation. En plus de la difficulté d'arriver à un consensus lors de la définition d'un tel langage, une autre difficulté est due aux problèmes liés à l'implémentation de systèmes de compilation efficaces supportant l'ensemble des fonctionnalités du langage.

Cet inconvénient peut être évité grâce à une autre approche, basée sur la notion de *mémoire virtuellement partagée*. Cette approche consiste à permettre l'utilisation du modèle de programmation des machines multiprocesseurs à mémoire partagée (la multiprogrammation) pour programmer des machines multiprocesseurs à mémoire distribuée, plus simples à construire, moins chères, plus facilement extensibles. Le programmeur garde la vision d'une mémoire globale commune uniformément accessible par tous les processeurs. La gestion des accès à cette mémoire est effectuée de manière transparente pour le programmeur par le système de mémoire virtuellement partagée. Ce système prend en charge le problème de la localisation des données et gère les accès à celles-ci, en générant éventuellement des messages de communication interprocesseurs lorsqu'il s'agit d'accès à une mémoire distante. La plupart des systèmes qui ont implémenté ce type d'approche se sont fixés pour objectif la portabilité : des codes écrits en vue d'une exécution sur des machines à mémoire *physiquement* partagée sont censé pouvoir s'exécuter également dans des environnements à mémoire *virtuellement* partagée, avec très peu de modifications (idéalement aucune).

Notre thèse se situe dans ce cadre : nous nous intéressons à l'utilisation d'un modèle de programmation pour les machines à mémoire *partagée* (le *multithreading*) sur des architectures physiquement *distribuées*, grâce à un système de *mémoire virtuellement partagée*. Dans la suite de ce chapitre, nous introduisons les concepts de base liés au multithreading et à la mémoire virtuellement partagée.

2.2 Parallélisme à grain fin sur architectures distribuées : le projet PM^2

Cette section introduit la notion de *thread*, discute l'utilisation du multithreading et présente en détail PM^2, l'environnement multithread pour architectures à mémoire distribuée qui a servi de cadre pour le travail présenté dans cette thèse.

2.2.1 Les processus légers

Le multithreading est une technique permettant de séparer d'une part la gestion des *flots d'exécution* d'un programme, et d'autre part les *ressources* fournies par le système d'exploitation pour réaliser cette exécution. Ces flots d'exécution, appelés *processus légers* ou *threads*, possèdent très peu de caractéristiques propres : une pile d'exécution et un compteur ordinal stocké dans un descripteur en première approximation. Ils partagent un certain nombre de ressources au sein de l'entité d'exécution de base du système d'exploitation, le *processus lourd*, traditionnellement appelé *processus*. Le code, l'espace d'adressage virtuel, la table des fichiers ouverts ou encore la table

de traitement des signaux sont des exemples de telles ressources qui sont propres à un processus lourd et que partagent tous les threads contenus par ce processus. Ces propriétés sont illustrées dans la figure 2.2.

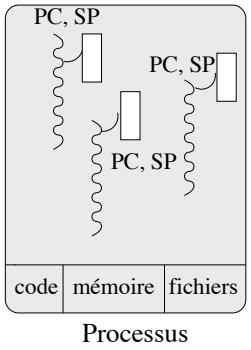


FIG. 2.2 – Les processus légers (*threads*) sont des flots d'exécution à l'intérieur d'un processus. Leurs caractéristiques propres se réduisent pratiquement à une pile (SP) et à un compteur ordinal (PC) de programme. Les ressources système du processus englobant, telles que le code, la mémoire et la table des fichiers, sont partagées.

La taille réduite des ressources propres aux threads s'exprime en kilooctets, alors que celle d'un processus s'exprime en mégaoctets. Ceci explique le fait que la gestion des threads est beaucoup plus efficace que celle des processus. Les opérations de création, changement de contexte, etc. sont généralement au moins un ordre de grandeur plus rapides. Ceci constitue un atout majeur en faveur de leur utilisation comme support des tâches parallèles dans le calcul à hautes performances. En particulier, elles constituent le support idéal pour les applications à parallélisme massif et à grain fin, comportant un grand nombre d'opérations de gestion de tâches (création, destruction, changement de contexte). Une utilisation fréquente des threads vise la réduction des périodes d'inactivité des processeurs grâce à un recouvrement des opérations d'entrée-sortie par du calcul : un thread bloqué en attente peut céder sa place à un autre thread prêt à s'exécuter.

Threads de niveau noyau et threads de niveau utilisateur

Les performances d'un environnement à base de threads dépendent du *niveau* d'implémentation de leur gestion. Cette gestion peut être implémentée soit au sein du noyau du système, soit en mode utilisateur. Dans le premier cas il s'agit de threads de niveau noyau (*kernel-level threads*) et dans le deuxième de threads de niveau utilisateur (*user-level threads*).

Lorsque l'ordonnancement est implémenté au niveau système, les opérations de création, destruction, commutation ou synchronisation nécessitent une intervention du noyau. Les descripteurs des threads et les structures de données de gestion sont implémentées dans l'espace d'adressage du noyau. Ceci permet une gestion correcte des appels bloquants et une répartition efficace des threads prêts sur les processeurs disponibles, puisque les threads sont ordonnancés par le noyau. Ainsi, l'ordonnancement au niveau noyau permet d'implémenter un parallélisme réel sur des machines multiprocesseurs à mémoire partagée (SMP). Ces avantages ont cependant une contrepartie : les opérations de gestion comportent un surcoût dû aux commutations entre les deux modes d'exécution (mode utilisateur et mode privilégié).

Lorsque l'ordonnancement est implémenté au niveau utilisateur, toutes les opérations de gestion sont plus efficaces, car elles se déroulent complètement en espace utilisateur, sans aucune intervention du noyau. En revanche, l'ordonnanceur de threads n'a pas la possibilité d'accéder aux processeurs de la machine sous-jacente. Par conséquent, il ne peut pas exploiter le parallélisme réel d'une machine multiprocesseurs. Il n'introduit qu'un pseudo-parallélisme, de la même manière qu'un ordonnanceur de processus UNIX sur une machine monoprocesseur. De plus, l'ordonnanceur de threads implémenté en espace utilisateur est ignoré par le noyau, qui ne voit qu'un seul processus (lourd). Dans cette situation, un éventuel appel bloquant d'un thread a pour effet le blocage du processus lourd conteneur et par conséquent de tous les autres threads de ce processus lourd. L'appel bloquant ne peut être donc pas recouvert par du calcul effectué par les autres threads du processus, comme cela était possible dans le cas des threads de niveau noyau.

Une solution à ce problème consiste à utiliser uniquement des primitives non-bloquantes, typiquement une fonction asynchrone de démarrage de l'opération et une fonction de test de fin d'opération. Un thread qui appelle la première peut aussi reprendre le contrôle dès que la requête est lancée et peut scruter périodiquement à l'aide de la deuxième pour tester si le résultat est disponible. Si ce n'est pas le cas, il a la possibilité de céder le processeur à un autre thread qui recouvre ainsi cette attente par un autre calcul. Cette solution, apparentée à l'attente active, peut cependant avoir des conséquence négatives sur les performances et, de plus, nécessite la réécriture des primitives bloquantes de la librairie standard. Une autre solution pourrait être construite en utilisant des fonctionnalités d'entrées-sorties asynchrones permettant à un processus d'être prévenu par un signal lors de la satisfaction d'une requête non-bloquante émise précédemment par un thread utilisateur de ce processus. Ces fonctionnalités ne sont pourtant pas proposées par tous les systèmes d'exploitation, ce qui limite la portabilité d'un ordonnanceur de threads utilisateur basé sur ce principe.

Enfin, certains systèmes comme Linux et Solaris supportent les deux types d'ordonnancement de threads et permettent une approche hybride. Deux niveaux d'ordonnancement se superposent dans ce cas : d'une part un ordonnancement de threads de niveau noyau effectué en mode privilégié, qui permet de mettre en œuvre un parallélisme réel et de réaliser un bon recouvrement en cas d'appels bloquants ; d'autre part, un ordonnancement efficace de threads utilisateur au dessus des threads de niveau noyau, qui apparaissent ainsi comme des processeurs virtuels accessibles en espace utilisateur.

Utilisations du multithreading

Les threads ont connu de multiples utilisations, non seulement pour des raisons de gains en performances, mais aussi pour le modèle qu'il fournissent pour l'expression du parallélisme.

Dans les systèmes d'exploitation. Les threads ont été utilisés de manière interne au noyau des systèmes pour prendre en charge efficacement les appels système [10] ou bien dans les systèmes de fichiers, pour améliorer le temps de réponse aux requêtes en permettant leur traitement de façon concurrente [78].

Dans les supports d'exécution des langages parallèles. Beaucoup de compilateurs de langages parallèles (Ada, CC++, HPF, OpenMP, etc.) utilisent des bibliothèques de threads comme cible de leur générateur de code. En plus des gains en efficacité, cette approche diminue la complexité du compilateur, puisqu'elle s'appuie sur l'utilisation de primitives élaborées de gestion d'activités parallèles. De plus, les bibliothèques de threads offrent une bonne

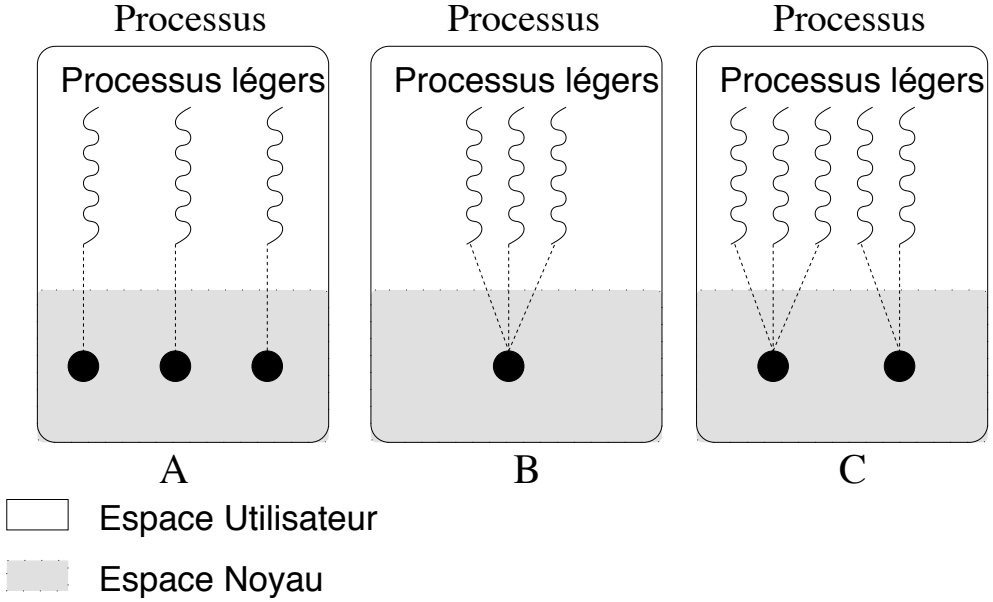


FIG. 2.3 – Dans la figure A, l'ordonnanceur de threads est implémenté au niveau système. Le noyau a donc connaissance des threads et gère le placement des threads concurrents sur les processeurs (parallélisme réel). Dans la figure B, l'ordonnanceur est implémenté au niveau utilisateur. Un seul processus léger peut alors être actif à la fois (pseudo-parallélisme), mais la gestion est plus efficace. Dans la figure C, l'ordonnancement est effectué à deux niveaux et peut combiner les avantages des deux approches précédentes.

portabilité sur plusieurs architectures. Une approche similaire est employée par des compilateurs de langages parallèles *distribués*, qui s'appuient non pas sur de simples bibliothèques de threads, mais sur des environnements multithreads distribués plus complexes qui intègrent des fonctionnalités de communication. Ainsi, le langage CC++ est compilé au dessus de Nexus [31], ou encore le langage HPF peut être compilé au-dessus de l'environnement PM^2 [80], grâce à une version modifiée du compilateur Adaptor [16].

Dans les applications. Les threads sont également utilisés directement au niveau des applications pour exprimer simplement et efficacement le parallélisme des tâches. Les threads sont généralement disponibles via une bibliothèque, mais certains langages comme Java intègrent même la notion de thread et proposent une classe d'objet Thread, pourvue de primitives de gestion. L'utilisation des threads convient naturellement à un modèle de programmation comportant des flots d'exécution qui partagent une mémoire commune, mais ils sont également utilisés dans des environnements distribuées, notamment pour recouvrir explicitement les temps de communication par du calcul.

La migration de threads

Une fonctionnalité proposée par plusieurs environnements multithread distribués est la migration de threads. Cette opération consiste essentiellement à déplacer le descripteur et la pile d'exécution d'un thread depuis un processus conteneur vers un autre, situé éventuellement sur

un nœud différent. Dans certains environnements (UPVM, PM^2), la migration comprend aussi le transfert de données allouées dynamiquement et utilisées par le processus léger. Nous allons détailler les problèmes liés à ce point dans la section 2.2.3.

De manière générale, la migration de processus légers a été utilisée à plusieurs fins :

- pour réaliser un équilibrage dynamique de la charge (dans UPVM [20], Millipede [48] et PM^2) ;
- pour "rapprocher" les processus légers des données qu'ils manipulent, de manière à réduire le coût des communications (dans Ariadne [69] et dans une certaine mesure dans Millipede). Ce principe a d'ailleurs motivé l'utilisation de la migration des *tâches* dans des environnements qui n'utilisent pas le multithreading. Jégou [49] a illustré une telle utilisation dans le contexte de la compilation d'applications à parallélisme de données.
- pour permettre de changer dynamiquement de support physique d'exécution, afin de ne pas dégrader la réactivité d'un nœud au cas où il est sollicité pour un traitement plus prioritaire [48]. Ceci est valable lorsque l'application parallèle s'exécute sur un réseau de machines partagées par plusieurs utilisateurs.

Suivant les motivations qui ont déterminé l'utilisation de la migration, différentes politiques ont été mises en œuvre quant à la migration des *données* manipulées par les processus légers. Ainsi, dans Ariadne [69], les threads migrent lorsqu'ils ont besoin d'accéder à des données globales situées sur un autre nœud. Après migration, ils continuent à s'exécuter sur le nouveau site jusqu'au moment où ils ont besoin de migrer de nouveau. La migration de threads est ici l'unique moyen d'accéder à des données distantes accessibles à travers une mémoire globale virtuellement partagée. Dans cet environnement, les seules données qui se déplacent sont les données locales stockées dans la pile du thread.

Dans Millipede [48, 33], la migration des threads est guidée par un module d'équilibrage de charge. Les données non locales (toujours globalement partagées) ne migrent pas immédiatement avec le thread. Elles sont migrées uniquement lorsque le thread y accède, de manière à améliorer la localité lors des accès futurs.

Dans PM^2, la migration de threads a été également conçue comme mécanisme d'équilibrage de charge, mais les threads ont la possibilité d'allouer dynamiquement des données privées, qui les suivent en cas de migration. Ces données ne sont pas accédées par d'autres threads, comme c'était le cas dans les systèmes à mémoire virtuellement partagée. Cette technique a été mise à profit dans le contexte d'utilisation de PM^2 comme cible de compilateurs de langages à parallélisme de données, où la parallélisation se réalise par partition de domaine. À chaque thread est associée une fraction de l'ensemble des données de l'application et c'est sur ce sous-ensemble que le thread effectue ses traitements.

2.2.2 L'environnement PM^2 : portabilité et efficacité

PM^2 (*Parallel Multithreaded Machine*) [80, 81] est un environnement multithread distribué développé à l'origine pour supporter de manière efficace et portable l'exécution d'applications irrégulières à parallélisme massif. Ces applications peuvent se décomposer en un grand nombre d'activités concurrentes et de durées différentes. Pour prendre en compte efficacement ce type d'applications, PM^2 repose sur une bibliothèque de threads de niveau utilisateur appelée MARCEL, qui fournit des primitives particulièrement performantes de gestion de threads. L'interaction

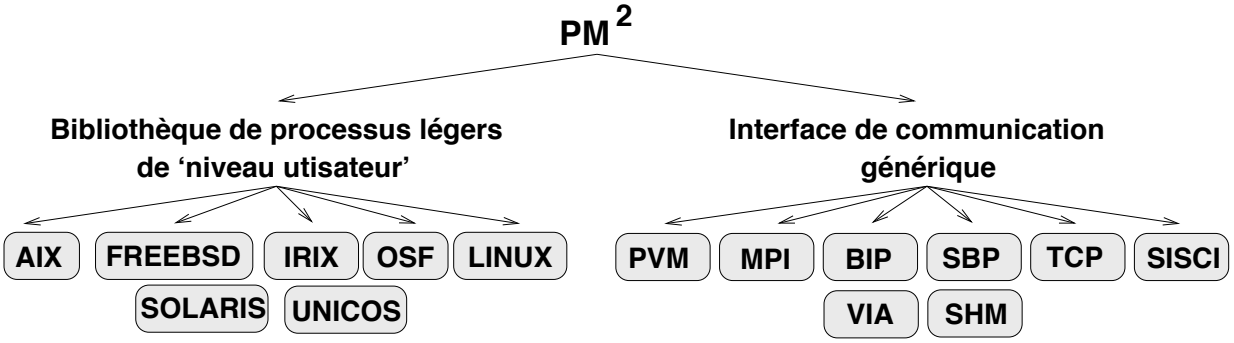


FIG. 2.4 – L'environnement multithread distribué PM^2 est basé sur deux composants logiciels : une bibliothèque de processus légers de niveau utilisateur et une interface de communication générique.

entre threads situés sur des nœuds différents se fait par un mécanisme d'*appel de procédure à distance* (*RPC, remote procedure call)* : un thread peut demander l'exécution d'un traitement à distance, qui peut être pris en charge par un thread de service préexistant ou par un nouveau thread. Ce mécanisme est implémenté au-dessus d'une couche de communication appelée MADELEINE, portable sur une grande variété d'interfaces et protocoles de communications. Enfin, toujours pour servir de support efficace à l'exécution des applications irrégulières, PM^2 propose la migration de threads comme mécanisme permettant d'implémenter différentes politiques de régulation de charge : les threads peuvent être migrés de manière préemptive transparente d'un nœud à un autre pour permettre un équilibrage efficace.

MARCEL. Les opérations de création, changement de contexte, etc. pouvant être très nombreuses dans les applications visées, l'implémentation efficace de ces opérations est cruciale et justifie l'implémentation d'une bibliothèque de threads de niveau utilisateur.

MARCEL fournit une interface de programmation basée sur la norme POSIX [43] semblable à celle de la bibliothèque *Pthreads* [74], mais introduit également des fonctionnalités liées à la migration de threads et à l'exploitation efficace des machines multiprocesseurs à mémoire partagée. Actuellement, MARCEL est disponible sur les architectures suivantes : Sparc, x86, Alpha, PowerPC, Mips et RS6000.

MADELEINE. La bibliothèque de communications de PM^2 a été développée pour assurer la portabilité de l'environnement sur un grand nombre d'interfaces de communication. Elle fournit une interface optimisée pour les opérations du type appel de procédures à distance au dessus de protocoles et bibliothèques "classiques" (TCP, MPI) ainsi que sur les interfaces de communications des réseaux hautes performances, comme BIP [88], VIA [26] ou SISCI [42]. Afin d'allier portabilité et efficacité, MADELEINE permet la transmission zéro-copie sur des réseaux à hautes performances comme Myrinet [14] ou SCI [42].

Bien que développée initialement pour PM^2, MADELEINE a évolué également vers une utilisation dans d'autres environnements. Ainsi, elle sert d'interface de communication pour des systèmes à mémoire virtuellement partagée (DSM-Threads [75], Mome [50, 51]) ou pour d'autres bibliothèques de communication, telles que MPI [72]. Actuellement, les travaux sur MADELEINE s'orientent vers le support des communications en contexte hétérogène.

	Changement de contexte	Création
SMP	0.500 μs	1.0
non-SMP	0.150	0.4

TAB. 2.1 – Performances de MARCEL (μs)

	SISCI/SCI	BIP/Myrinet
Latence (μs)	4	7
Bande passante (Mo/s)	80	125

TAB. 2.2 – Performances de MADELEINE

Performances

Les coûts des opérations élémentaires de MARCEL (changement de contexte, création de thread) sont présentés dans la table 2.1. Ces chiffres correspondent à des performances de crête, atteignables pourtant assez souvent en pratique. Les mesures ont été effectuées sur une machine à 4 processeurs PIII 550 MHz sous Linux 2.4.

Les performances de MADELEINE sont illustrées dans la table 2.2. L'objectif de cette bibliothèque est d'assurer une implémentation efficace de son interface générique sur des bibliothèques de communication hautes performances. Le surcoût est donc faible (environ quelques microsecondes) par rapport aux performances de ces bibliothèques.

Le tableau 2.3 présente le temps de migration pour un thread de petite taille (1 kilooctet). Ces courbes ont été obtenues en mesurant les temps d'aller-retours d'un processus léger entre deux nœuds.

PM^2 : un support exécutif flexible

Conçu selon une approche modulaire, en gardant comme objectifs premiers la portabilité et l'efficacité, PM^2 propose au programmeur un environnement de programmation parallèle riche en fonctionnalités, mais aussi particulièrement flexible. En effet, chaque mécanisme intégré dans PM^2 propose des options ou des modes de fonctionnement alternatifs et peut être configuré par le programmeur en vue de l'exécution optimale d'une application donnée. Ainsi, MADELEINE propose plusieurs modes de transmission des données avec différents compromis entre garanties pour le programmeur et efficacité, ainsi que plusieurs types d'interaction avec MARCEL pour la scrutation du réseau. Marcel fournit également plusieurs types d'ordonnancement des threads ainsi que plusieurs modes de fonctionnement, dont certains proposent des mécanismes spécifiques aux architectures à base de multiprocesseurs à mémoire partagée. De plus, tous les composants intègrent

SISCI/SCI PII 450 MHz	BIP/Myrinet PII 450 MHz	TCP/Myrinet PII 450 MHz	TCP/Fast Ethernet PII 450 MHz
62	75	280	373

TAB. 2.3 – Coût de la migration d'un thread (en μs).

des fonctionnalités de traçage et de profilage qui s'avèrent extrêmement utiles à la mise au point et à l'optimisation des performances des applications PM^2. Toutes ces fonctionnalités font de PM^2 un bon support exécutif pour des classes variées d'applications parallèles.

2.2.3 Allocation et migration iso-adresse

Une fonctionnalité centrale dans la conception de PM^2 est la migration de threads. Ce mécanisme, utilisable via la primitive `pm2_migrate (marcel_t tid, int pid)`, migre le processus léger `tid` vers le processus lourd `pid`. Cette fonction peut être appelée par le processus léger lui-même (migration non-préemptive) ou par un autre (migration préemptive).

Déroulement d'une migration

La migration se déroule essentiellement en trois étapes (figure 2.5). Le thread est d'abord suspendu par MARCEL et son descripteur ainsi que la partie utilisée de la pile sont empaquetés dans un tampon de communication. Ce tampon est ensuite transféré par le réseau vers le processus destinataire. Enfin, un espace mémoire est allouée sur ce nœud, pour permettre le dépaquetage du descripteur et de la pile et ensuite le thread reprend son exécution.

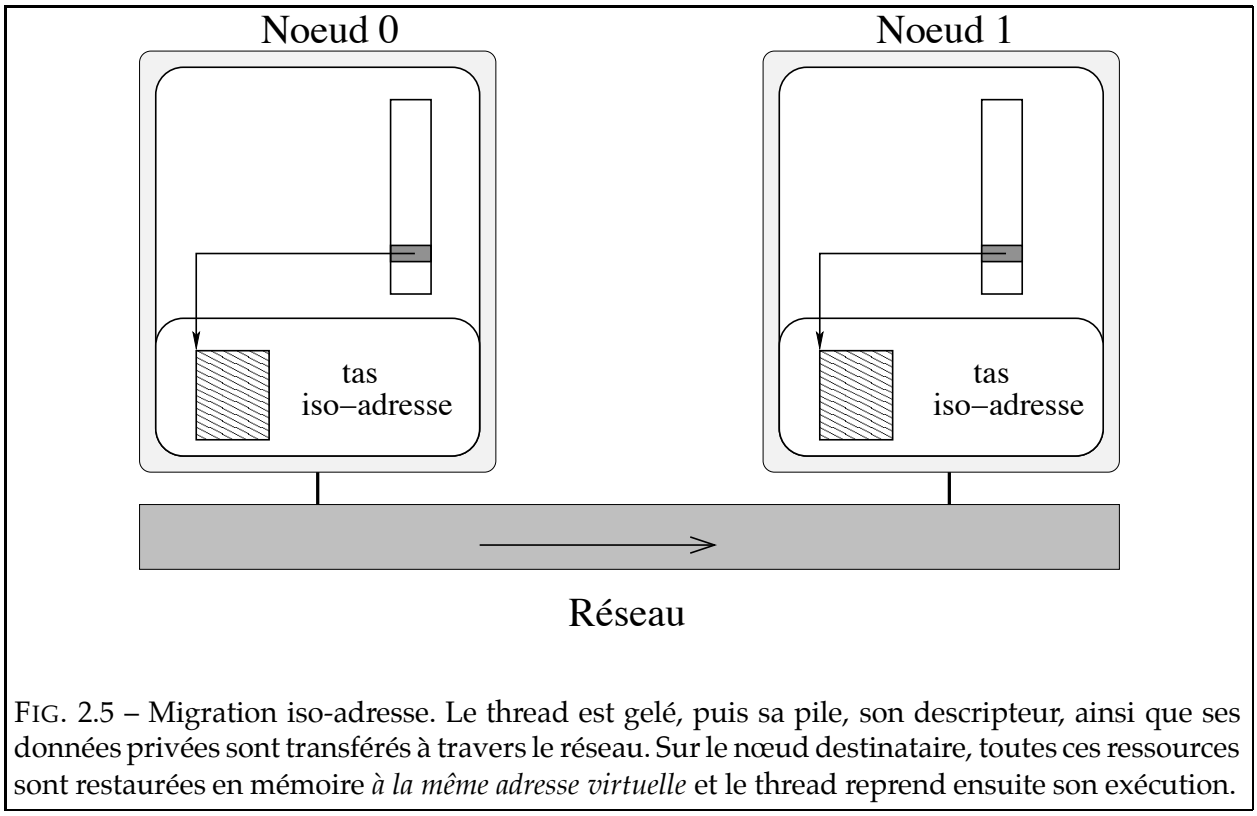


FIG. 2.5 – Migration iso-adresse. Le thread est gelé, puis sa pile, son descripteur, ainsi que ses données privées sont transférés à travers le réseau. Sur le nœud destinataire, toutes ces ressources sont restaurées en mémoire *à la même adresse virtuelle* et le thread reprend ensuite son exécution.

PM^2 laisse au programmeur la possibilité de configurer le processus de migration et d'insérer d'éventuels traitements spécifiques à trois points précis du déroulement d'une migration, par l'intermédiaire de trois fonctions qui doivent être déclarées à PM^2. La première est appelée sur le nœud de départ juste après l'empaquetage de la pile et permet d'empaqueter d'autres données de

niveau utilisateur qui doivent migrer avec le thread. Symétriquement, la deuxième est appelée sur le nœud destinataire et permet le dépaquetage de ces données immédiatement après le dépaquetage de la pile. Enfin, la troisième est appelée juste avant le réveil du thread et permet d'exécuter des traitements post-migration. Ces fonctions peuvent être utilisées lors de l'implémentation d'entités abstraites de plus haut niveau par des threads migrables, lorsque la migration de telles entités nécessite des traitements pré- ou post-migration.

Limitations

Il n'est pas toujours possible de migrer un thread, notamment lorsqu'il utilise des ressources propres au processus englobant, comme par exemple des descripteurs de fichiers, ou, plus généralement, quand il effectue des entrées-sorties. Chaque thread a un état de migrabilité qui peut être changé implicitement (de manière temporaire) par PM^2 (par exemple pendant l'exécution d'un RPC), mais peut aussi (et doit !) être modifié explicitement par le programmeur pour empêcher la migration du thread lors d'un appel système.

Une restriction plus importante concerne l'hypothèse d'homogénéité faite par PM^2 : l'architecture sous-jacente doit comporter des nœuds identiques (même type de processeur et même système d'exploitation) et la migration doit s'effectuer entre processus identiques, avec le même code binaire et la même structure de l'espace d'adressage virtuel.

Migration en présence de pointeurs

Lorsque la pile d'un thread migre d'un processus à un autre, un problème se pose quant à la cohérence de certaines références situées dans la pile, si celle-ci est relogée à une adresse (virtuelle) différente de celle d'origine. Il s'agit tout d'abord du chaînage des contextes dans la pile (par *frame pointers*) généré par le compilateur, mais aussi des pointeurs de niveau applicatif qui référencent des données stockées également dans la pile (données locales). La méthode proposée dans Ariadne [69] consiste à identifier et translater les références locales. Si les pointeurs utilisés pour le chaînage des contextes dans la pile (*frame pointers*) peuvent être identifiés en connaissant le mécanisme par lequel le compilateur les génère, cette identification devient problématique dans le cas des pointeurs utilisateur. Une solution possible à ce problème a été mise en œuvre dans les premières versions de l'environnement PM^2 : les pointeurs de niveau applicatif qui référencent des données locales étaient déclarés explicitement au système, de manière à permettre leur mise à jour en cas de migration. Cette méthode a pourtant le désavantage d'alourdir considérablement l'écriture du code.

Un problème plus délicat survient lorsqu'un thread manipule des références à des données situées en dehors de sa pile, typiquement dans des zones allouées dynamiquement dans le tas du processus englobant à l'aide de la primitive `malloc` du langage C. Dans ce cas aussi il est envisageable de prévoir une mise à jour post-migration, à condition que les pointeurs soient déclarés aux système. Cette approche ne convient évidemment pas aux applications complexes qui utilisent un grand nombre de pointeurs et, de plus, ne peut assurer la cohérence des éventuels pointeurs générés par le compilateur pour optimiser l'accès aux données.

Une bien meilleure solution à ce problème consiste à assurer la cohérence de ces références en relogeant les données en question sur le nœud destinataire à une adresse virtuelle identique à celle d'origine. C'est ce que nous appelons une approche *iso-adresse*.

Allocation iso-adresse

Cette technique repose sur la capacité d'allouer de la mémoire *localement*, tout en gardant une cohérence *globale* : le mécanisme d'allocation garantit que tout intervalle d'adresses virtuelles alloué sur un nœud est inutilisé sur tous les autres nœuds. Un processus léger qui utilise un ensemble de tels intervalles peut alors être migré et relogé avec ses ressources sur un autre nœud, tout en préservant les adresses virtuelles utilisées (*allocation et migration iso-adresse*). Après recopie, l'exécution peut reprendre, car aucun traitement post-migration n'est nécessaire : tous les pointeurs restent valides, grâce à la migration iso-adresse.

La mise en place de ce mécanisme dans PM^2 rend possible la migration *préemptive* de threads et son utilisation au sein de régulateurs dynamiques de charge qui agissent de manière *transparente* sur les threads de l'application, sans coopération de leur part. La conception et la mise en œuvre de l'allocateur iso-adresse de PM^2 a fait l'objet de nos travaux dans une première partie de cette thèse. Sa validation par intégration dans des régulateurs de charge au sein de deux compilateurs de langages parallèles (C* et HPF) a été faite en collaboration avec Christian Pérez.

Le principal inconvénient de la stratégie iso-adresse est que la taille de la mémoire globalement disponible est de l'ordre de la taille de la mémoire virtuelle d'un seul nœud (en pratique, approx. 3,5 Go, pour un espace d'adressage 32 bits). Ce problème perdra de son importance lors du passage à un espace d'adressage 64 bits.

2.2.4 Utilisations de la plate-forme PM^2

En dehors de son utilisation par programmation directe, PM^2 a également servi de cible pour des compilateurs de langages parallèles. En particulier, deux compilateurs de langages à parallélisme de données pour des architectures distribuées ont été adaptés pour générer du code multithread PM^2 : un compilateur HPF (Adaptor) et un compilateur C* (UNH-C*).

Dans les langages à parallélisme de données, la parallélisation se fait par partition du domaine : les processeurs abstraits du langage exécutent en parallèle les mêmes opérations sur des données distinctes. Ce modèle d'exécution applique une méthode appelée *Owner Computes Rule* : les calculs ont lieu dans le processus où les résultats des expressions doivent être stockés. La distribution des données contrôle donc la distribution des calculs. Dans ce contexte, l'équilibrage de charge s'obtient classiquement grâce à une distribution équilibrée des données. Dans le cas des applications irrégulières, une régulation dynamique peut être réalisée par *redistribution* des données, spécifiée explicitement à des points précis du programme. Une approche différente a été proposée dans [15] et consiste à découpler le code HPF [41] d'une application des traitements propres à la régulation de charge. Le principe consiste à projeter les processeurs abstraits de HPF sur des threads et à utiliser un module de régulation de charge (externe à l'application) qui migre les threads de manière *transparente* : les threads sont déplacés à leur insu et continuent à s'exécuter indépendamment de leur nœud d'accueil. Cette solution a été implémentée en utilisant les threads PM^2 et repose sur le mécanisme de migration préemptive des threads fourni par cet environnement. La redistribution des processeurs abstraits constitue une application directe de la migration iso-adresse des threads *avec leurs données privées*.

2.2.5 Limitations de la plate-forme PM^2

L'objectif premier du projet PM^2 a été de fournir un environnement de programmation multithread pour des architectures *distribuées* telles que les grappes de PC. À partir du modèle de programmation multithread, à l'origine défini pour les machines à mémoire physiquement partagée, PM^2 propose une interface de programmation multithread qui exploite des architectures à mémoire distribuée. Cette prise en compte de la distribution est explicite : au niveau du modèle de programmation de PM^2, les différents nœuds de la configuration sont visibles et exploités par des communications explicites, à travers les appels de procédure à distance. Le programmeur doit donc gérer lui-même les aspects liés à la localisation des données et des traitements.

Pourtant, un travail important reste à accomplir pour l'exploitation plus simple des architectures distribuées, un objectif intéressant étant de pouvoir programmer une telle architecture sans avoir à prendre en compte sa nature distribuée ! La *transparence* des aspects liés à la distribution et à la localisation des entités manipulées par le programme est une propriété importante dans ce cadre. Un pas dans cette direction a été fait par l'introduction de la migration *préemptive* et *transparente* des threads dans PM^2, avec des données attachées : un thread peut changer de nœud pendant son exécution sans s'en apercevoir tout en accédant les mêmes données (privées). Ceci introduit une forme intermédiaire de partage, le *partage des traitements*. Un autre pas décisif reste à faire pour atteindre l'objectif de la transparence : mettre en œuvre un mécanisme permettant le *partage des données* : les variables d'une applications doivent être uniformément accessibles par tous les threads d'une application multithread, quelle que soit la localisation des threads et celle des données. Ce rôle est rempli par un système de type *mémoire virtuellement partagée*. La construction d'un tel système pour PM^2 a fait l'objet principal de notre travail de thèse.

2.3 Mémoire virtuellement partagée (MVP)

Dans cette section, nous introduisons la notion de *mémoire virtuellement partagée* et nous rappelons les principales problématiques associées à la conception de ce type de système. Nous présentons également quelques évolutions récentes dans ce domaine [44].

Comme nous l'avons expliqué dans la section précédente, le concept de *mémoire virtuellement partagée* (MVP), en anglais *distributed shared memory* (DSM)[1] permet l'utilisation du modèle de programmation des machines multiprocesseurs à mémoire partagée pour programmer des machines multiprocesseurs à mémoire distribuée. Le programmeur utilise l'abstraction d'une mémoire globale, dont la gestion est effectuée de manière transparente par le système, qui cache les détails liés à la localisation physique des données. Dans ce qui suit, nous présentons les problématiques les plus communes dans la conception des systèmes à mémoire virtuellement partagée ainsi que les solutions proposées par les systèmes existants [83, 87].

2.3.1 Modèles de cohérence

Le *modèle de cohérence* d'une MVP définit la sémantique des accès aux données partagées en spécifiant des contraintes sur l'ordre dans lequel les accès effectuées par un processeur à l'ensemble ou à une partie de la mémoire partagée doivent être visibles aux autres processeurs. Le

[1] Plusieurs traductions en français ont été proposées : mémoire distribuée virtuellement partagée, mémoire partagée répartie, etc. Tout au long de ce manuscrit, nous utiliserons le syntagme : mémoire virtuellement partagée.

modèle de cohérence comporte un ensemble de règles qui spécifient le comportement de la MVP et les garanties qu'elle fournit au programmeur. Ces garanties peuvent être plus ou moins fortes et le choix d'un modèle a des conséquences en termes de facilité de programmation et de coût des accès. Plus les garanties sont fortes, plus la latence des accès mémoire est élevée, alors que la programmation s'en trouve simplifiée. A l'opposé, les modèles qui proposent des garanties plus faibles arrivent à améliorer la latence des accès grâce à des techniques telles que le réordonnancement ou l'agrégation des accès, et à une cohérence limitée à un sous-ensemble des données partagées et garantie uniquement à des moments bien spécifiés. En contrepartie, leur utilisation nécessite un effort de programmation plus important.

Dans la suite du manuscrit, nous appelons *protocole de cohérence* toute implémentation particulière d'un *modèle de cohérence*. Un modèle peut être implémenté par plusieurs protocoles, en utilisant des algorithmes et des supports d'implémentation (c'est-à-dire mécanismes logiciels ou matériels) différents. Pour être valide, tout protocole doit satisfaire les contraintes spécifiées par le modèle associé.

On sépare traditionnellement les modèles de cohérence en deux catégories : les *cohérences fortes* et les *cohérences faibles*. Le critère de séparation utilisé par les différentes classifications existantes n'est pas toujours le même. Nous avons choisi d'appeler *cohérences faibles* les modèles qui font la distinction entre les opérations ordinaires sur la mémoire partagée (lectures et écritures) et les opérations de synchronisation (blocage / libération des verrous, barrières) en spécifiant des contraintes différentes pour chacun de ces types d'opérations. Les *cohérences fortes* ne font pas cette différenciation au niveau des contraintes, qui sont les mêmes pour toutes les opérations sur la mémoire partagée.

La suite de cette sous-section présente les principales caractéristiques des modèles de cohérence les plus représentatifs.

Les cohérences fortes

Cohérence stricte. Extension naturelle de la notion de cohérence de la plupart des mémoires des machines monoprocesseurs, ce modèle exige que la valeur retournée par toute lecture corresponde à la valeur écrite à la même adresse par l'opération d'écriture la plus récente. La cohérence stricte n'est pas appliquée en pratique dans les MVP, car elle serait trop coûteuse, puisqu'elle impliquerait des synchronisations globales à chaque accès à la mémoire partagée.

Cohérence séquentielle. Un modèle moins restrictif, largement utilisé par les premières implémentations de systèmes à MVP, a été proposé par Lamport [57] : la cohérence séquentielle. Une mémoire est séquentiellement cohérente si le résultat de toute exécution est le même que si les opérations de tous les processeurs sur la mémoire étaient exécutées dans un ordre séquentiel dans lequel les opérations de chaque processeur apparaissent dans l'ordre du programme qu'il exécute. Autrement dit, lorsque les processus de l'application s'exécutent en parallèle sur plusieurs nœuds (ou même sur un seul nœud, sur un système à temps partagé), tout entrelacement des séquences d'accès des différents processus est valide, mais tous les processus doivent "voir" le même ordre total des accès à la mémoire. La cohérence séquentielle est proposée par la plupart des machines multiprocesseurs à mémoire partagée, mais aussi par un grand nombre de systèmes MVP (Ivy, Mirage, etc.).

Cohérence PRAM et cohérence processeur. La *cohérence PRAM* (Pipelined Random Access Memory) admet que les différents processeurs ne "voient" pas nécessairement les opérations

sur la mémoire dans le même ordre, mais exige que les écritures effectuées par un processeur soient vues par tous les autres processeurs dans le même ordre que celui dans lequel elles ont été émises (*pipeline*). Une cohérence très proche de la cohérence PRAM est la *cohérence processeur*. Les deux ont souvent été confondues, car les différences sont subtiles : la cohérence processeur exige en plus que tous les processeurs voient toutes les écritures *à la même adresse mémoire* dans le même ordre. PLUS [13], Merlin [68], RMS [66] sont quelques exemples de MVP qui utilisent la cohérence processeur.

Les cohérences faibles

Cohérence faible. Bien que la cohérence processeur permette une meilleure efficacité que les cohérences plus fortes, elle est encore trop restrictive pour beaucoup d'applications, car souvent il n'est pas utile que les écritures d'un nœud soient vues dans le même ordre sur tous les autres nœuds. En effet, lorsque la même donnée partagée est écrite plusieurs fois dans une section critique, il est inutile de propager aux autres nœuds toutes les valeurs intermédiaires de la donnée avant la sortie de la section critique. Une meilleure approche consiste à propager la valeur finale uniquement à la sortie de la section critique. Ceci peut être réalisé en introduisant des *variables de synchronisation* et en y associant des actions qui assurent la cohérence. À la fin d'une synchronisation, toutes les modifications effectuées localement sont propagées vers les autres nœuds et, inversement, la mémoire locale prend en compte les modifications effectuées par les autres nœuds. Ce modèle de cohérence (appelé *cohérence faible*) distingue les *accès ordinaires* à la mémoire (lectures et écritures) des *accès de synchronisation* (c'est-à-dire les accès aux objets de synchronisation, tels que les barrières) et limite les exigences de cohérence séquentielle uniquement aux accès de synchronisation. En plus, un accès de synchronisation doit attendre que *toutes* les requêtes d'accès ordinaire le précédant soient satisfaites, alors que les accès ordinaires doivent attendre *seulement* la satisfaction des accès de synchronisation précédents. Une variante de ce modèle est utilisée dans les machines SPARC de Sun Microsystems.

Cohérence à la libération. Une limitation du modèle de cohérence faible vient du fait que la MVP ne sait pas si un accès de synchronisation est effectué parce qu'un processus a fini d'écrire une donnée ou parce qu'il se prépare à la lire. Par conséquent, le système doit agir pour les deux cas et garantir d'une part que toutes les modifications locales ont été propagées aux autres nœuds et d'autre part que les modifications effectuées par les autres nœuds sont visibles sur le nœud local. Le modèle de *cohérence à la libération* (en anglais *release consistency*[2], dont le nom fait référence à la libération d'un verrou lors de la sortie de la section critique) distingue deux types d'accès de synchronisation, pour permettre au programmeur de spécifier si le processus va rentrer dans une section critique (*acquire*) ou s'il vient d'en quitter une (*release*). Le contrat entre le programmeur et la mémoire spécifie que lorsque le programme effectue un *acquire* la mémoire actualise les valeurs des données partagées modifiées sur les autres nœuds. Lors d'un *release*, les modifications locales sont propagées aux autres nœuds. Si tous les accès aux données partagées sont protégés par des sections critiques, la mémoire garantit une sémantique équivalente à celle garantie par la cohérence séquentielle, tout en étant plus efficace (figure 2.6). Les synchronisations se font généralement à l'aide des verrous et des barrières. L'acquisition (respectivement la libération) d'un verrou correspond à une opération *acquire* (respectivement *release*). Arriver à une barrière correspond à un *release*,

[2]Une autre traduction moins exacte souvent utilisée en français est *cohérence relâchée*.

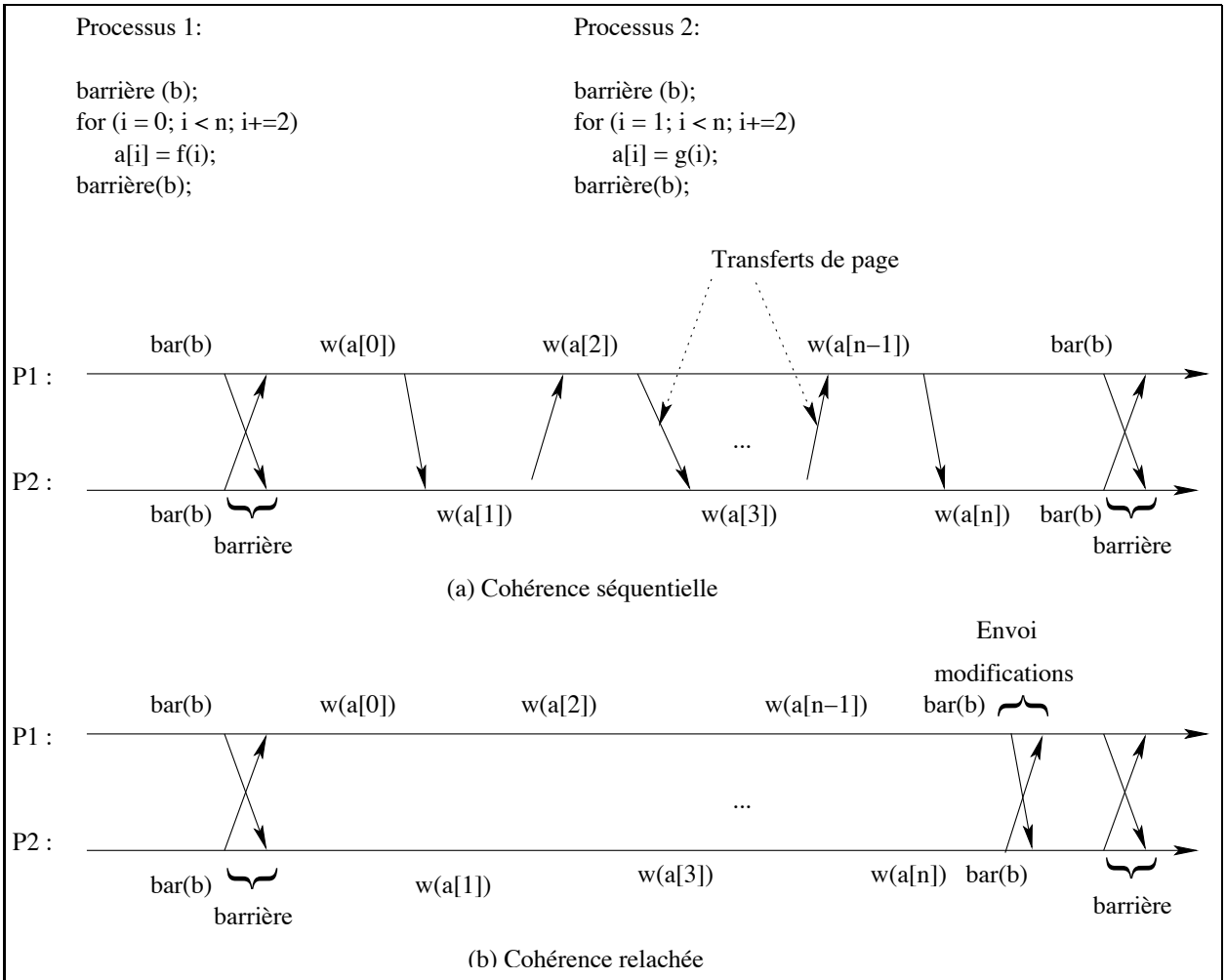


FIG. 2.6 – Cohérence à la libération versus cohérence séquentielle.

La cohérence à la libération peut générer moins de messages que la cohérence séquentielle tout en garantissant la même sémantique du programme. Dans cet exemple, deux processus placés sur des nœuds différents modifient en parallèle les éléments d'un tableau (le premier les éléments d'indice pair, le deuxième les éléments d'indice impair). Les processus se synchronisent par barrière avant et après cette mise à jour. La barrière est implémentée par un échange total. Dans le cas de la cohérence séquentielle, les écritures concurrentes peuvent générer des aller-retours de la page entre les deux nœuds à cause du faux partage, alors que la cohérence à la libération permet des implémentations qui évitent les aller-retours, en autorisant les écrivains multiples. Dans ce dernier cas, les modifications effectuées par un nœud sont transmises à l'autre nœud lors de la deuxième barrière (ou plus tard, dans le cas d'une implémentation paresseuse).

alors que la quitter correspond à un *acquire*.

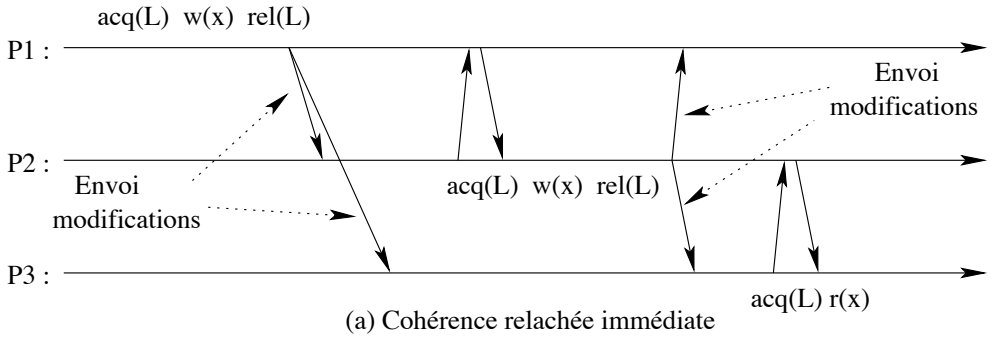
Ce modèle de cohérence peut être implémenté suivant deux stratégies. Si les modifications effectuées par un nœud à des données partagées sont systématiquement propagées aux autres nœuds lors d'une opération *release*, on parle de *cohérence à la libération en mode immédiat* (*eager release consistency*). Étant donné qu'une telle action n'est pas nécessaire avant que les

données soient effectivement accédées, une autre stratégie consiste à reporter cette propagation au prochain *acquire* effectué par l'un des autres nœuds, qui signale ainsi son intention d'accéder les données partagées (*cohérence à la libération en mode paresseux*, en anglais *lazy release consistency*). Cela a pour effet de réduire les communications, car les modifications sont envoyées uniquement aux nœuds qui en ont besoin (figure 2.7). En contrepartie, la consommation mémoire est beaucoup plus importante que dans le cas des stratégies immédiates. En effet, dans ce dernier cas, la mémoire où sont stockées les modifications peut être libérée tout de suite après l'opération *release*, alors que les stratégies paresseuses nécessitent que ces modifications soitent gardées plus longtemps (par exemple, jusqu'à ce que tous les nœuds les obtiennent par un *acquire*). Le moment où la mémoire correspondante peut être libérée est généralement difficile à détecter. Une nouvelle classe de stratégies qui résolvent ce problème est représentée par les protocoles *à référence* (*home-based*). Ces stratégies seront discutées dans la section 2.3.4.

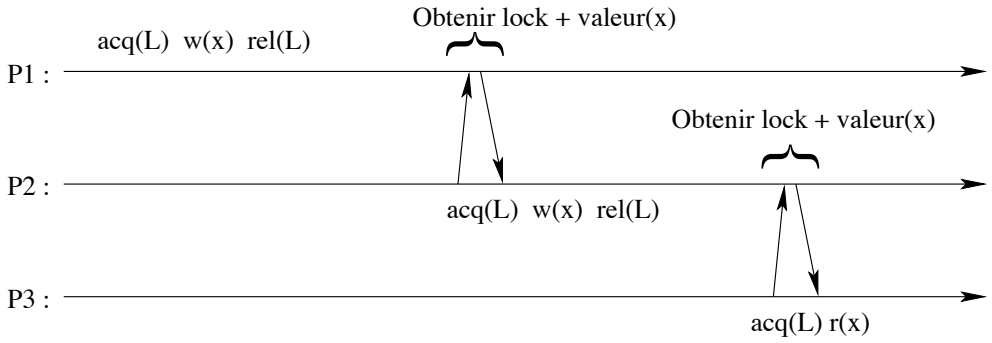
Par ailleurs, la cohérence à la libération est l'un des premiers modèles à populariser les *écrivains multiples* : plusieurs nœuds peuvent avoir accès en écriture à la même page de mémoire partagée en même temps et modifier des données différentes, sans que cela génère des allers-retours de la page entre les nœuds (phénomène dû au *faux partage*). Les mécanismes mis en œuvre sont relativement complexes. Avant d'accéder une page, chaque écrivain en sauvegarde une copie (*jumelle*). Plus tard, il détecte ses modifications sur la page en comparant sa page courante à la jumelle sauvegardée et les stocke dans une structure de données appelée *différence*. Ces modifications peuvent être envoyées lors d'un *release* à tous les nœuds qui ont une copié de la page (stratégie immédiate) ou bien plus tard, uniquement aux nœuds qui effectuent un *acquire* (stratégie paresseuse). Dans les deux cas il s'agit d'une approche de type *mise à jour*. Une alternative consiste à envoyer uniquement des *notifications d'écriture* à la place des *différences*, permettant au nœuds concernés d'invalider localement ces pages modifiées (approche de type *invalidation*). Par la suite, l'accès à une page invalidée déclenche alors la recherche des *différences* associées à la page et stockées sur d'autres nœuds. Ces différences sont transférées en mémoire locale et appliquées localement afin de reconstituer la cohérence de la page avant de permettre l'accès au nœud local. La cohérence à la libération en mode immédiat a été illustrée dans Munin [19] et la cohérence à la libération en mode paresseux dans TreadMarks [4]. Les deux systèmes permettent l'utilisation des écrivains multiples.

Cohérence d'entrée. Un modèle de cohérence qui réduit encore plus les communications engendrées et qui réduit également le faux partage est la *cohérence d'entrée* (*entry consistency*). Ce modèle exige que chaque variable partagée soit associée à un verrou ou à une barrière qui protège les accès à la variable. Lors d'une opération *acquire* sur un objet de synchronisation en entrée de section critique, uniquement les données associées à cet objet sont mises à jour, d'où le nom du modèle. Cette association permet de minimiser les communications au prix d'une plus grande complexité du programme. Les accès à un verrou peuvent être en mode *exclusif* ou en mode *non-exclusif*. Pour modifier les données associées à un verrou, un processus doit obtenir le verrou en mode exclusif. En revanche, plusieurs processus peuvent acquérir le verrou en mode non-exclusif et lire simultanément les données associées.

Cohérence de portée. Le modèle de cohérence le plus récent est la *cohérence de portée* [45] (*scope consistency*). Ce modèle se propose de fournir l'efficacité de la cohérence d'entrée tout en gardant la simplicité de programmation de la cohérence à la libération. L'idée est d'épargner au programmeur l'effort d'associer explicitement les données partagées aux objets de synchronisation, mais de fournir des mécanismes pour que les objets à mettre à jour lors d'un *acquire* soient déterminés implicitement par la MVP. Le modèle repose sur la notion de



(a) Cohérence relachée immédiate



(a) Cohérence relachée paraisseuse

FIG. 2.7 – Cohérence à la libération en mode immédiat versus cohérence à la libération en mode paresseux.

La cohérence à la libération en mode paresseux peut générer encore moins de messages que la cohérence à la libération en mode immédiat. Lors d'un *release*, aucune action de cohérence n'est entreprise par la stratégie paresseuse, alors que la stratégie immédiate envoie des nouvelles valeurs des données modifiées à tous les nœuds qui en possèdent une copie. Seuls les processeurs qui effectuent un *acquire* par la suite sont informés des modifications et peuvent éventuellement les obtenir par dans le même message, comme illustré sur la figure (approche de type mise à jour). Une approche alternative consiste à envoyer uniquement des notifications d'écriture (et non pas les modifications mêmes). Ces modifications sont alors obtenues plus tard, lorsque les données sont effectivement accédées.

portée : chaque entrée dans une nouvelle section critique *ouvre* une nouvelle portée, qui est *fermée* par la sortie de section critique correspondante. Toutes les sections critiques protégées par le même verrou correspondent à la même portée. Les modifications apportées à des données partagées à l'intérieur d'une portée seront rendues visibles aux processus qui ouvrent des nouvelles sessions de la même portée. Les données à mettre à jour sont identifiées implicitement par la MVP (figure 2.8). Ce modèle de cohérence a été implémenté par Brazos [98] et JIAJIA [100].

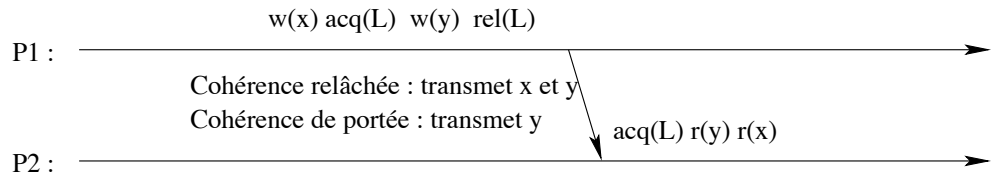


FIG. 2.8 – Cohérence de portée versus cohérence à la libération.

Dans cet exemple, le processus `P1` écrit `x`, effectue un *acquire* sur le verrou `L`, ensuite écrit `y` et finalement relâche le verrou. Le processus `P2` obtient le verrou et ensuite lit `y` et `x`. La cohérence à la libération garantit que le `P2` lit les valeurs de `x` et `y` écrites par `P1`, alors que la cohérence de portée propage uniquement la modification de `y` et n'offre aucune garantie sur la valeur de `x` lue par `P2`, qui a été modifié par `P1` en dehors de la portée définie par `L`.

2.3.2 Choix de conception

Mis à part le modèle de cohérence, d'autres facteurs déterminent les performances globales des systèmes à mémoire virtuellement partagée. En particulier, il s'agit de la granularité du partage, des stratégies algorithmiques, ainsi que le mode de gestion de la localisation des données.

Granularité du partage

La granularité de l'unité de partage détermine la taille des blocs de données manipulés par les protocoles de cohérence. Ce paramètre a un effet important sur l'efficacité globale du système. De manière générale, les systèmes à MVP matérielle utilisent une granularité fine fixée (souvent la taille d'une ligne du cache). Les MVP de niveau logiciel utilisent souvent la page comme unité (fixe) du partage, lorsqu'elles reposent sur le système de mémoire virtuelle. Par contre, quand elles sont implémentées au niveau du langage et utilisent un compilateur spécifique, les MVP logicielles proposent une granularité de taille *variable* dans l'espace mémoire : la taille de l'objet partagé.

Lorsque les applications sont caractérisées par une bonne localité spatiale, l'utilisation d'une granularité large permet parfois de réduire le nombre de messages de communication engendrés par les protocoles de cohérence. En effet, lorsque des objets voisins en mémoire, situés par exemple sur la même page, sont accédés séquentiellement par un processeur, un seul transfert de page suffit pour satisfaire plusieurs accès, par un effet de *prefetching*. D'autre part, plus l'unité est large, plus elle peut contenir des données partagées et donc plus il est probable que cette page soit demandée plus souvent par différents processeurs sur différents nœuds, même s'ils accèdent à des données différentes (*faux partage*). Les granularités fines réduisent le risque de faux partage, mais nécessitent plus d'espace mémoire pour les structures de données de gestion, car il y a plus d'unités à gérer.

Stratégies algorithmiques d'implémentation : réplication et migration

Les algorithmes utilisés pour implémenter les MVP abordent essentiellement deux problématiques : (1) l'allocation et la répartition des données partagées de manière à minimiser le coût des accès ; (2) le maintien d'une vue cohérente des données partagées sur chaque processeur qui y

accède, tout en minimisant le surcoût de cette gestion. Deux techniques sont largement utilisées : la *réplication* et la *migration*. La réplication est utilisée pour permettre les accès concurrents de plusieurs nœuds à la même donnée, ou à des données différentes situées dans la même unité de partage. La plupart des systèmes appliquent la réplication en lecture (*read-replication*), lorsque les lectures concurrentes sont prédominantes, mais cette technique peut également être utilisée pour des accès en écriture (protocoles à écrivains multiples). Dans ce dernier cas, des actions complémentaires sont nécessaires pour assurer la cohérence des différentes copies modifiées de manière concurrente. Alternativement, la migration peut être utilisée pour gérer les accès concurrents en écriture, sans faire recours à la réplication : la donnée migre d'un nœud à l'autre et seul le nœud qui la possède couramment a le droit de la modifier. Suivant le type de réplication utilisé, les algorithmes d'implémentation des MVP se classifient en trois catégories.

SRSW (single reader, single writer). Ce type d'algorithme interdit toute réplication. Les accès distants se traduisent alors systématiquement soit par des transferts des valeurs entre les nœuds, soit par des migrations des données. Bien que simples, ces algorithmes ont une efficacité généralement limitée, car aucune parallélisation des accès concurrents aux données sur des nœuds différents n'est possible.

MRSW (multiple reader, single writer). Utilisé dans beaucoup de MVP, ce type d'algorithme est basé sur la réplication en lecture et permet à plusieurs nœuds d'accéder en parallèle la même donnée par des accès locaux. Parmi les copies répliquées, une seule est modifiable. Lorsqu'un accès en écriture à cette copie est détecté, le protocole de cohérence doit généralement empêcher les autres copies d'être modifiées (stratégie de type *invalidation*).

MRMW (multiple reader, multiple writer). Lorsque les accès concurrents en écriture sont dominants, on peut permettre également leur parallélisation, en autorisant l'existence de plusieurs copies consultables et modifiables sur des nœuds différents (*full-replication*). Afin de maintenir la cohérence, les modifications effectuées sont diffusées aux autres copies ou à une copie de référence, dans les algorithmes *à référence* (en anglais : *home-based*). Contrairement aux algorithmes MRSW basés sur l'invalidation, on applique généralement ici une stratégie de type *mise à jour*.

Gestion de la localisation et des accès aux données

Lorsqu'on accède à une donnée partagée, la MVP doit garantir que cette donnée est à jour, conformément au modèle de cohérence implémenté. Dans le cas des protocoles à écrivain unique (algorithme SRSW ou MRSW), ceci repose sur un mécanisme permettant de localiser le nœud qui détient la copie à jour, pour ensuite transférer cette copie en mémoire locale. Li et Hudak [64] ont proposé plusieurs algorithmes MRSW qui implémentent de tels mécanismes. Dans tous ces algorithmes, la copie à jour est toujours détenue par le nœud (unique à tout moment) qui est autorisé à modifier la donnée. Avant de les discuter, nous devons introduire les notions suivantes :

propriétaire : le nœud ayant écrit le dernier sur la page ;

gestionnaire : le nœud qui connaît le propriétaire d'une donnée et qui est chargé de gérer les accès en écriture à cette donnée ;

ensemble des copies : l'ensemble des nœuds qui possèdent des copies d'une donnée partagée répliquée.

Le propriétaire d'une donnée partagée peut être *fixe* ou *dynamique*. Dans le premier cas, la donnée est toujours possédée par le même nœud, qui est le seul à pouvoir y accéder en écriture. Tous les autres nœuds doivent négocier avec le propriétaire à chaque fois qu'ils ont besoin de modifier la donnée. C'est une approche coûteuse, qui n'est pas utilisée en pratique. Les implémentations existantes reposent sur des approches à propriété dynamique : le propriétaire d'une donnée peut changer durant l'exécution du programme. Les gestionnaires utilisés dans ce cas sont de deux types : *centralisés* et *distribués*. Les gestionnaires distribuées peuvent être *fixes* ou *dynamiques*.

Gestionnaire centralisé. Toutes les requêtes de lecture ou écriture sont adressées au gestionnaire, qui est le seul nœud à connaître le propriétaire d'une donnée partagée. Le gestionnaire fait suivre la requête au propriétaire et attend que le nœud ayant généré la requête confirme la réception de la donnée. Lors d'une écriture, c'est encore le gestionnaire qui envoie des invalidations à l'ensemble des copies.

Gestionnaire centralisé amélioré. Cet algorithme est similaire à la précédente, mais introduit une décentralisation qui améliore les performances de la MVP : c'est le propriétaire, et non pas le gestionnaire, qui connaît l'ensemble des copies. Cet ensemble est transmis avec la donnée au nouveau propriétaire, qui se charge de l'envoi des invalidations avant d'accéder la donnée.

Gestionnaire distribué fixe. Un pas supplémentaire vers la décentralisation est introduit en distribuant le gestionnaire : chaque nœud est gestionnaire d'un sous-ensemble des données partagées. Le gestionnaire d'une donnée est toujours le même est peut être déterminé en utilisant une fonction de hachage quelconque.

Gestionnaire distribué à diffusion. Dans cet algorithme, il n'y a pas de gestionnaire physique : l'identité du propriétaire d'une donnée partagée est déterminée en diffusant une requête à tous les nœuds. Comme dans les deux algorithme précédents, c'est le propriétaire qui connaît l'ensemble des copies. L'inconvénient de cette approche est que tous les nœuds doivent traiter toutes les diffusions, ce qui diminue les performances globales.

Gestionnaire distribué dynamique. Le gestionnaire distribuée dynamique est similaire au gestionnaire distribué fixe, mais il a une particularités : tous les nœuds gardent l'identité du propriétaire pour *toutes* les données partagées (et non pas pour un sous-ensemble), mais cette identité n'est pas sûre (*propriétaire probable*). Toutes les requêtes sont acheminées vers le propriétaire probable, qui le plus souvent est le vrai propriétaire. Néanmoins, si ce n'est pas le cas, ce propriétaire présumé fait suivre la requête au nœud qui représente le propriétaire probable conformément à ses informations locales. La mise à jour des informations concernant les propriétaires probables se fait lors de la réception des requêtes et des messages d'invalidation, afin de minimiser le nombre de messages nécessaires pour identifier le vrai propriétaire. Li et Hudak [64] montrent que l'algorithme permet toujours de trouver le propriétaire et que le nombre maximum de messages nécessaires pour cette identification varie logarithmiquement avec le nombre de processeurs.

Les algorithmes décrits ci-dessus ont été conçus pour des protocoles MRSW, mais peuvent être facilement adaptés à des protocoles SRSW, qui n'utilisent pas d'ensemble des copies puisqu'il n'y a pas de réplication, et donc ne nécessitent pas d'invalidation non plus. Néanmoins, comme nous l'avons déjà souligné, les algorithmes SRSW sont rarement utilisés en pratique.

Dans le cas des protocoles à écrivains multiples (MRMW), la copie "la plus récente" peut éventuellement devoir être reconstituée à partir de modifications éparpillées sur plusieurs nœuds. C'est le cas dans les protocoles qui implémentent la *cohérence à la libération* suivant une stratégie

paresseuse. Ces protocoles utilisent ce que nous pouvons appeler un *gestionnaire distribué dynamique paresseux* : l'identité des nœuds qui doivent transmettre leurs modifications est déterminée dynamiquement en utilisant des informations partielles détenues par plusieurs nœuds. Ces protocoles intègrent un mécanisme de gestion des écritures concurrentes par *différences* : lors de la reconstitution d'une page "à jour", les nœuds autorisés à écrire sur la page transmettent uniquement les modifications effectuées et non pas toute la page. Dans d'autres approches MRMW, un nœud parmi ceux qui ont l'autorisation d'écriture sur la donnée est chargé de maintenir à jour une copie de la donnée. Il s'agit des algorithmes *à référence* (*home-based*), dans lesquels la copie de référence permet d'éviter l'utilisation des mécanismes sophistiqués de reconstitution que nous venons de mentionner.

2.3.3 Implémentations

Le niveau d'implémentation représente l'un des choix les plus importants lors de la conception d'une MVP, car il a une influence directe sur le coût et sur les performances du système. De manière générale, on distingue trois types d'implémentations.

Des implémentations au niveau matériel. Supérieures en efficacité, mais plus coûteuses, ces implémentations comportent des mécanismes *matériels* de localisation et/ou de réplication automatique des données dans les mémoires des différents processeurs, de manière transparente pour les couches logicielles. Souvent, des techniques matérielles sophistiquées sont utilisées pour maintenir la cohérence et minimiser la latences des accès, ce qui fait que ces implémentations sont utilisées dans les machines de haut de gamme, lorsque la performance est plus importante que le coût. Les architectures à MVP matérielle se divisent essentiellement en trois catégories : cc-NUMA (*cache-coherent nonuniform memory architectures*), COMA (*cache-only memory architecture*) et RMS (*reflexive memory system*). Les systèmes cc-NUMA distribuent statiquement un espace d'adressage global sur les différentes mémoires locales des processeurs, qui reste accessibles à la fois par le(s) processeur(s) du nœud local et par ceux des autres nœuds, avec des latences d'accès différentes. L'implémentation de la MVP est basée sur des protocoles d'invalidation et peut supporter des modèles de cohérence faible. Des exemples représentatifs sont Memnet [23], Dash [60] et SCI [40]. Les systèmes COMA utilisent les mémoires locales comme d'énormes caches pour les données (*attraction memories*). La répartition des données est ici dynamique s'adapte au comportement de l'application : les données se déplacent sur le nœud où elles sont utilisées. KSR1 [32] et DDM [39] sont deux exemples de ce type d'architecture. Enfin, les systèmes RMS reposent sur un mécanisme matériel de réplication immédiate des données à l'écriture (*mirror memory*). Les accès en lecture sont toujours efficaces, car ils se font toujours en mémoire locale. L'approche est illustrée par RMS [66] et Merlin [68].

Des implémentations au niveau logiciel. Les MVP de niveau logiciel se proposent de fournir l'abstraction d'une mémoire partagée sur des machines faiblement couplées sans que cela nécessite du support matériel spécifique. On distingue : des MVP implémentées au niveau du système d'exploitation, dans le noyau (Mirage [29]) ou au-dessus de celui-ci (Clouds [89]) ; des MVP sous forme de bibliothèques en espace utilisateur (TreadMarks [4], JIAJIA [100], CVM [52]) dont certaines nécessitent des modifications du système d'exploitation (Ivy [63], Munin [19]), généralement mineures ; des implémentations au niveau du langage de programmation, nécessitant un compilateur spécifique (Linda [1], Orca [9]). Les MVP implémentées sous forme de bibliothèques ou dans le système d'exploitation intègrent

Système	Architecture, SE, Communication	Implémentation	Type d'algorithme	Cohérence	Granularité	Stratégie
Ivy [63]	stations de travail Apollo + Aegis ; token ring 12 Mb/s	Bibliothèque + SE modifié	MRSW	SEQ	1 ko	INV
Mermaid [104]	stations de travail Sun + SunOS, DEC Firefly + TAOS ; TCP, UDP	Bibliothèque + SE modifié	MRSW	SEQ	1 ko, 8 ko	INV
Munin [19]	stations de travail Sun 3 + System V ; Ethernet	bibliothèque + SE modifié + relieur + exécutif + préprocesseur	SRSW, MRSW, MRMW	REL	variable (objets)	INV, MAJ
Blizzard [94]	bibliothèque de comm. Tempest	Bibliothèque + SE modifié	MRSW	SEQ	32-128 octets	INV
Mirage [29]	Vax 11/750, Ethernet	noyau SE modifié	MRSW	SEQ	ligne cache (512 octets)	INV
Clouds [89]	Sun 3/50 et Sun 3/60 + UNIX	niveau SE, au-dessus du noyau	MRSW	SEQ, INC	8 ko	SPE
Gobelins [65]	ix86 + Linux ; Gigabit Ethernet, Myrinet	niveau SE + modification SE	MRSW	SEQ	page	INV

TAB. 2.4 – Systèmes MVP implémentés au niveau logiciel *avec modification du système d'exploitation*. SEQ : cohérence séquentielle ; REL : cohérence à la libération ; REL(P) : cohérence à la libération en mode paresseux ; PRO : cohérence processeur ; ENT : cohérence d'entrée ; POR : cohérence de portée ; INC : incohérent ; INV : invalidation ; MAJ : mise à jour ; DEP : dépend de divers critères ; SPE : stratégie spéciale, ad-hoc.

souvent le mécanisme de gestion de la cohérence avec le système de mémoire virtuellement partagée : lors d'un accès à une donnée absente de la mémoire locale, un défaut de page déclenche la localisation de la page sur un autre nœud et son transfert en mémoire locale. Les principales caractéristiques de quelques MVP logiciels représentatifs sont présentées dans les tables 2.4 (pour les systèmes implémentés au niveau du système d'exploitation ou ayant nécessité une modification de celui-ci) et 2.5 (pour les systèmes implémentés complètement en espace utilisateur).

Bien que moins performantes que les MVP matérielles, les MVP de niveau logiciel sont plus flexibles et permettent une meilleure adaptation des mécanismes de cohérence aux besoins des applications exécutées. Certaines proposent plusieurs modèles de cohérence, d'autres proposent plusieurs protocoles associés au même modèle. Un exemple de ce type est Midway [11], qui fournit la cohérence processeur, la cohérence à la libération et la cohérence d'entrée. D'autres systèmes proposent plusieurs protocoles associés au même modèle de cohérence. Dans Munin [19], par exemple, qui fournit une cohérence à la libération, le programmeur peut associer un protocole différent à chaque donnée partagée, en fonction du type d'accès.

Des implémentations hybrides. Une solution au problème du compromis performance/facilité de programmation est proposée par les systèmes MVP *hybrides*, qui combinent des mécanismes logiciels et des mécanismes matériels afin d'assurer la cohérence. Une approche de ce type, illustrée par PLUS [13] consiste à fournir la réplication ou la migration des données partagées au niveau logiciel, alors que la cohérence des données répliquées est assurée

Système	Architecture, SE, Communication	Implémentation	Type d'algorithme	Cohérence	Granularité	Stratégie
Midway [11]	DECstation 5000 + Mach 3.0 ; Ethernet 10 Mb/s, ATM 155 Mb/s	exécutif + compilateur	MRMW	ENT, REL, PRO	variable (objets)	MAJ
TreadMarks [4]	DECstation 5000/240 ; Ethernet 10Mb/s, ATM 100Mb/s	bibliothèque	MRMW	REL(P)	page	MAJ, INV
Linda [1]	MicroVax, Intel iPSC hypercube ; Ethernet	langage	MRSW	SEQ	variable (tuples)	DEP
Orca [9]	arbitraire	langage	MRSW	DEP	variable (objets)	MAJ
Quarks [54]	SunOS 4.1, HP-UX	bibliothèque	MRSW	SEQ, REL	région, page	INV, MAJ
Millipede [48]	Windows NT ; Myrinet	bibliothèque	MRSW	SEQ	adaptable	INV
CVM [52]	Sparc + Solaris 2.5, Dec Alpha et IBM SP2 + AIX 4.1 ; UDP/IP	bibliothèque	MRSW, MRMW	SEQ, REL(P)	page	INV
Shasta [93]	Alpha Workstation ; Memory Channel, ATM	support compilateur	MRSW	SEQ	variable	INV
Cashmere-2L [99]	DEC AlphaServer, Memory Channel	langage	MRMW	REL(P) (home)	page	MAJ, INV
DSM-Threads [75]	ix86 + Linux, Sparc + SunOS 4.1.x ; TCP/Ethernet 10 Mb/s	bibliothèque + threads	SRSW, MRSW, MRMW	SEQ, REL, ENT	page	INV
Brazos [98]	Compaq Proliant 1500, Windows NT 4.0 ; FastEthernet 100 Mb/s	bibliothèque + exécutif + threads	MRMW	POR	page	MAJ
JIAJIA [100]	stations de travail Sun Sparc + Solaris 2.4, IBM-SP2 et Dawning 1000A + AIX 4.1, + Linux 2.0 ; UDP/IP, Ethernet	bibliothèque	MRMW	POR	page	INV, MAJ
Mome [50]	Sparc + Solaris 2.4, NEC Cenju-3 ou Cenju-4 + Mach 3.0, ix86 + Linux 2.0 ; TCP/IP, MPI, SCI	bibliothèque	MRSW, MRMW	SEQ, REL	page	MAJ
CLRC [7]	Celeron et K6 + Linux ; UDP/IP/Fast Ethernet 100 Mb/s	bibliothèque	MRMW	REL (P)	page	MAJ, INV
DOSMOS [90]	ix86 + Linux 2.0 ; PVM et MPI / Fast Ethernet 100 Mb/s, MPI/BIP/Myrinet	bibliothèque + langage	MRMW	REL	variable	MAJ, INV

TAB. 2.5 – Systèmes MVP implémentés au niveau logiciel *en espace utilisateur*. SEQ : cohérence séquentielle ; REL : cohérence à la libération ; REL(P) : cohérence à la libération en mode paresseux ; PRO : cohérence processeur ; ENT : cohérence d'entrée ; POR : cohérence de portée ; INC : incohérent ; INV : invalidation ; MAJ : mise à jour ; DEP : dépend de divers critères ; SPE : stratégie spéciale, ad-hoc.

par des mécanismes matériels. Dans FLASH [56], l'exécution des actions du protocole de cohérence est prise en charge par un processeur spécialisé. D'autres systèmes utilisent des accélérateurs matériels pour le systèmes de mémoire virtuelle (Galactica Net [101]). Dans SHRIMP [45], un mécanisme matériel de mise à jour reporte automatiquement sur la copie de référence d'une donnée répliquée toutes les modifications effectuées sur d'autres copies.

2.3.4 Performance des MVP : progrès récents et évolutions actuelles

L'idée de construire un espace d'adressage virtuel global sur une architecture de machines faiblement couplées à mémoire distribuée a été proposée il y a une quinzaine d'années [62]. Intéressant pour le programmeur, auquel il offre la possibilité de tirer profit de la puissance de calcul des machines distribuées de manière transparente en proposant un modèle de programmation simple, le concept de MVP s'est heurté dès le début au problème de la performance. Le défi de la plupart des efforts de recherche dans le domaine des MVP a été de proposer des mécanismes permettant d'approcher les performances obtenues en utilisant une programmation par passage explicite de messages.

L'efficacité des systèmes à MVP est limitée par plusieurs facteurs. L'utilisation d'une granularité de partage supérieure à la taille des données individuelles (généralement une page) engendre des *communications inutiles*. Ceci est surtout pénalisant dans les protocoles qui implémentent des modèles de cohérence forte, tels que la cohérence séquentielle, basés sur des algorithmes à écrivain unique (SRSW ou MRSW), car seulement une partie des données situées la même page sont utilisées. D'autre part, les accès concurrents de plusieurs nœuds à des données différentes situées sur la même page créent une situation de *faux partage* qui déclenche des aller-retours coûteux de la page entre les différents nœuds, en augmentant encore plus le trafic des données. Les déclenchements des défauts de pages à l'intérieur des sections critiques représentent une autre source potentielle d'inefficacité, car ils ont pour effet de *dilater les sections critiques* et de renforcer les sérialisations. Enfin, les MVP de niveau logiciel souffrent en plus des *surcoûts liés aux protocoles*, dus à l'exécution des actions de cohérence par les processeurs des nœuds qui doivent interrompre l'exécution des applications pour répondre aux différentes requêtes engendrées par les protocoles.

Les efforts qui ont contribué à l'amélioration significative des performances des MVP ont été faits dans plusieurs directions. Une avancée importante est due aux modèles de cohérence faible, et tout particulièrement à la cohérence à la libération, introduite au début des années 1990. L'efficacité a été encore améliorée grâce aux protocoles à écrivains multiples (MRMW) qui ont fourni une solution au problème du faux partage. Les stratégies paresseuses ont ensuite permis de réduire encore plus les communications engendrées par les protocoles, à l'aide de mécanismes plus complexes. Un niveau de performance comparable a été par ailleurs obtenu grâce aux protocoles *à référence* (*home-based*), plus simples à mettre en œuvre. D'autres contributions ont visé la classification des schémas d'accès aux données et ont produit des systèmes multiprotocoles, dans lequel le programmeur peut associer des protocoles différents à des données différentes, suivant le type d'accès. Certaines MVP proposent même des protocoles adaptatifs, qui changent dynamiquement de stratégie en fonction du comportement de l'application. Enfin, récemment les MVP intègrent les bénéfices du multithreading, essentiellement pour permettre le recouvrement des communications par des calculs. Dans la suite de cette section, nous allons présenter les progrès récents dans ces différentes directions.

Stratégies paresseuses

Utilisées par beaucoup de protocoles de cohérence faible, les stratégies paresseuses peuvent concerner plusieurs étapes d'un protocole de cohérence : la propagation des opérations de cohérence aux autres nœuds (par des notifications d'écriture, qui précisent *quelles* données ont été modifiées), l'application de ces actions sur ces nœuds ; ainsi que la propagation des *valeurs* des données modifiées. Toutes ces opérations peuvent être effectuées de manière immédiate ou paresseuse et le choix d'une stratégie ou d'une autre peut avoir un impact important sur la performances des protocoles. De manière générale, les approches paresseuses permettent de réduire les communications au prix d'une complexité plus importante des mécanismes sous-jacents mis en œuvre.

Propagation de la cohérence. L'implémentation la plus simple du modèle de cohérence à la libération consiste à propager les notifications d'écriture à tous les nœuds qui possèdent des copies des pages modifiées, immédiatement après les opérations *release*. Une fois envoyées, ces informations deviennent obsolètes et peuvent donc être déchargées de la mémoire. Alternativement, ces notifications peuvent être envoyées plus tard, lors d'un prochain *acquire*, sélectivement au nœud qui effectue cet *acquire* (propagation paresseuse). Ceci a deux conséquences. D'une part, les informations sur les données modifiées ne peuvent plus être déchargées après le *release*, car elles pourraient être nécessaires lors d'un prochain *acquire*. D'autre part, le nœud ayant effectué un *release* doit fournir au nœud qui effectue un *acquire* toutes les notifications d'écriture que ce nœud n'a pas encore obtenues, dues éventuellement aux écritures des nœuds tiers. Par conséquent, même les notifications "anciennes" prises en compte localement ne peuvent pas être déchargées (jusqu'à une prochaine synchronisation globale), car d'autres nœuds pourraient encore en avoir besoin. La consommation mémoire exigée est par conséquent importante. De plus, un problème particulièrement difficile auxquel se sont heurtées la plupart des implémentations est celui de la détection du moment où ces notifications peuvent être déchargées en toute sécurité. La mise en œuvre de ce type de protocoles paresseux (illustrée dans TreadMarks [4]) repose sur un protocole d'estampillage permettant d'identifier précisément les notifications d'écriture à envoyer dans chaque cas.

Prise en compte des informations de cohérence. Même lorsque les notifications d'écriture sont envoyées immédiatement lors d'un *release*, les nœuds les ayant reçues peuvent les appliquer tout de suite, ou bien les stocker et les appliquer plus tard, lors d'un prochain *acquire*. Le *cohérence à la libération en mode immédiat* correspond à une propagation immédiate lors du *release*, suivie d'une application immédiate. La propagation immédiate associée à une application paresseuse est utilisée par un autre type d'implémentation (*delayed consistency* [25]). Un protocole MRMW de ce type est utilisé dans Cashmere [99]. Enfin, la *cohérence à la libération en mode paresseux* correspond traditionnellement à une propagation paresseuse suivie d'une application immédiate. Les deux sont déclenchées par l'opération *acquire* qui suit un *release*.

Propagation des données modifiées. Comme les notifications d'écriture, les valeurs des données modifiées peuvent à leur tour être transmises de manière immédiate ou paresseuse. En revanche, *l'application* de ces modifications sur les nœuds les ayant demandées est toujours immédiate. Dans les protocoles à écrivain unique, il existe toujours une seule instance modifiable de chaque unité de partage (page). Dans le modèle de cohérence à la libération, cette instance (toujours à jour) peut coexister temporairement avec d'autres copies en lecture seule, jusqu'au moment ou les nœuds qui possèdent ces copies se synchronisent avec le pro-

priétaire courant de la page. Lors de la synchronisation, les notifications d'écriture peuvent être transmises de manière immédiate ou paresseuse (cf. ci-dessus), et détermine l'invalidation des copies en lecture seule. Lors d'un accès en écriture à une copie invalidée, toute la page migre vers le nœud ayant généré l'accès. Le problème de la propagation paresseuse des données ne se pose donc pas.

Dans les protocoles à écrivains multiples, plusieurs nœuds peuvent modifier en même temps la même page. Lors d'un *release*, chaque écrivain détermine les modifications qu'il a effectuées sur la page depuis la dernière synchronisation et les stocke sous forme de *différences*. Ces différences qui peuvent ensuite être propagées aux autres nœuds de manière plus ou moins paresseuse. La forme la plus immédiate consiste à les envoyer lors du *release*, en même temps que les notifications d'écriture (protocole de type *mise à jour*). La forme la plus paresseuse utilisée, mise en œuvre dans TreadMarks [4], consiste à obtenir les différences uniquement au moment où on en a besoin, c'est-à-dire sur un défaut de page, qui détermine la recherche des différences sur un ou plusieurs nœuds, suivie de leur transfert et de leur application en mémoire locale. Conceptuellement, même la création des différences pourrait d'ailleurs être reportée jusqu'à ce qu'elles soient demandées pour la première fois !

Stratégies algorithmiques *à référence*

Une forme intermédiaire de propagation paresseuse des données modifiées lorsque plusieurs écrivains coexistent consiste à utiliser une transmission en deux étapes. Pour chaque page partagée on choisit un *noeud de référence* (*home node*) et les modifications sont propagées et appliquées de manière immédiate vers ce nœud uniquement, lors d'un *release*. Cela peut être fait soit en utilisant des *différences*, soit via un support matériel spécifique [105]. Les autres copies des pages modifiées sont invalidées par des notifications d'écriture. Le nœud de référence peut accéder à la page sans générer des défauts de page et sans créer de différences. Les autres nœuds obtiennent *la page entière* sur un défaut de page, depuis le nœud de référence. La propagation des différences vers le nœud de référence peut éventuellement être reportée au prochain *acquire*, mais pas plus, car les différences doivent rejoindre le nœud de référence avant que la page correspondante ne soit demandée par le nœud ayant effectué cet *acquire*. Les différences deviennent alors obsolètes et peuvent être déchargées de la mémoire.

L'utilisation d'une stratégie à référence a plusieurs avantages, comparativement aux stratégies traditionnelles d'implémentation de la cohérence à la libération en mode paresseux : (1) réduction du nombre de messages engendrés, car un défaut de page déclenche seulement une requête vers le nœud de référence, qui renvoie la page entière ; (2) simplicité, car les différences sont appliquées une seule fois, sur le nœud de référence ; elles ont donc une courte durée de vie et n'ont pas besoin des structures de gestion complexes gourmandes en espace mémoire, ni de protocoles d'estampillage ; (3) les écritures sur le nœud de référence n'engendrent pas de création de différences. Les désavantages potentiels sont : (1) le transfert d'une page entière sur un défaut de page, même s'il suffirait de mettre à jour un seul mot ; (2) les éventuels goulots d'étranglement qui peuvent apparaître si les nœuds de référence ne sont pas "bien" répartis.

Des protocoles à référence ont été implémentés pour la cohérence à la libération et pour la cohérence de portée, soit entièrement au niveau logiciel, soit en utilisant un mécanisme matériel de mise à jour automatique du nœud de référence (protocoles AURC : *Automatic Update Release Consistency* et AU-based ScC : *Automatic Update-based Scope Consistency* sur le système SHRIMP [45]). Actuellement, les stratégies à référence sont de plus en plus utilisées [105, 45],

car les mesures de performance effectuées ont montré dans un grand nombre de cas représentatifs une efficacité améliorée, comparativement aux stratégies purement paresseuses, plus complexes à mettre en œuvre et nécessitant plus d'espace mémoire.

MVP multiprotocoles et protocoles adaptatifs

Plusieurs études de performance sur des systèmes MVP ont montré qu'il n'existe pas de protocole de cohérence unique, qui soit le plus efficace quelle que soit l'application exécutée. Généralement, un protocole est meilleur qu'un autre pour un type d'accès donné. Au sein de la même application, les différentes données peuvent être accédées différemment et même le type d'accès à la même donnée peut changer pendant l'exécution, ce qui rend difficile toute comparaison globale des protocoles.

Ce problème a été abordé par Munin [19], qui a le mérite de proposer la première MVP multiprotocole : les différentes variables partagées sont annotées par le programmeur afin de spécifier du schéma d'accès (*read-only*, *migratory*, *write-shared*, *producer-consumer*, etc.). Le système utilise ces annotations pour associer aux données le protocole le plus efficace en fonction du schéma d'accès indiqué. Par ailleurs, le programmeur a la possibilité de paramétrer les protocoles afin d'améliorer leur efficacité. Tous les protocoles implémentent le même modèle de cohérence : la *cohérence à la libération*. La flexibilité offerte par Munin a pour contrepartie une plus grande complexité de la programmation, due aux annotations explicites. D'autre part, une limitation de cette approche vient du caractère *statique* des annotations, qui déterminent l'association d'un protocole à une donnée pour toute la durée d'exécution. Ceci qui ne convient cependant pas aux applications plus irrégulières dans lesquelles le schéma d'accès à la même donnée peut changer *dynamiquement*.

Une réponse aux limitations statiques mentionnées plus haut est apportée par les protocoles *adaptatifs*, qui changent dynamiquement de stratégie de cohérence suivant l'évolution des schémas d'accès dans le temps. Les protocoles de ce type (expérimentés dans TreadMarks [5, 3]) peuvent décider automatiquement plusieurs aspects : *écrivain unique* vs *écrivains multiples*, *invalidation* vs. *mise à jour*, agrégation des pages en unités éventuellement plus larges, etc.

Les deux approches présentées ci-dessus reposent sur une classification *statique* des schémas d'accès. Cette classification est figée par les annotations au niveau du langage dans le premier cas. Dans le deuxième cas, elle est figée au niveau de l'implémentation des critères de décision dans les protocoles adaptatifs. Des recherches sur la classification sont encore en cours actuellement [22] qui pourraient éventuellement remettre en cause les choix faits par les systèmes mentionnés.

Intégration du multithreading

L'efficacité des systèmes à MVP peut encore être améliorée grâce au *multithreading*. Cette technique a été souvent utilisée dans les environnements de programmation parallèles distribuée pour recouvrir les communications par des calculs et cette fonction a été également mise à profit dans les MVP récentes [98, 52]. Une présentation plus détaillée des raisons qui ont motivé l'intégration des threads dans les MVP sera faite dans la section 2.4, qui présentera également quelques systèmes ayant effectué cette intégration.

2.3.5 Modèles hiérarchiques et cohérence multi-niveaux

L'intérêt récent pour l'utilisation des plates-formes multi-grappes comme support pour le calcul hautes performances à grande échelle a mis en évidence l'importance de certaines problématiques jusqu'ici souvent négligées dans le domaine des systèmes à mémoire virtuellement partagée, telles que la tolérance aux fautes où l'hétérogénéité des architectures. Par ailleurs, les implémentations des modèles de cohérence classiques visent généralement des architectures "plates", qui supposent que les coûts des communications sont du même ordre pour toute paire de processeurs sous-jacents. Or, cette hypothèse n'est plus valable dans le cadre d'une architecture multi-grappes : les communications intra-grappes sont généralement plus rapides que les communications inter-grappes. La conception de protocoles de cohérence adaptés à ce type d'architecture est l'une des directions de recherche les plus importantes actuellement dans le domaine des MVP [7, 6].

2.3.6 Un défi : une plate-forme d'expérimentation pour MVP

Lorsqu'on considère l'ensemble des efforts de recherche dans le domaine des systèmes à mémoire virtuellement partagée, un constat s'impose : souvent, afin d'expérimenter un nouveau concept (modèle de cohérence, protocole, stratégie algorithmique, etc.), il a fallu construire un système complet. La plupart des composants du système reposent sur des mécanismes classiques, déjà implémentés précédemment, et seulement peu de composants sont conçus spécifiquement pour illustrer le concept proposé (nouveau modèle de cohérence, etc.). Pourtant, la validation expérimentale du concept n'est possible que si le système *complet* est fonctionnel.

Cette situation a deux conséquences immédiates. Tout d'abord, un investissement important en temps et ressources matérielles et humaines est nécessaire lors de la validation de nouvelles techniques, car cette validation nécessite à chaque fois la mise en œuvre d'un ensemble de mécanismes complexes de base, en plus de l'implémentation de la technique que l'on souhaite illustrer. Ceci a pour effet de rendre difficiles et longues les étapes d'implémentation et validation expérimentale. Une deuxième conséquence est que seule une comparaison des systèmes *complets* vis-à-vis d'un ensemble d'applications fixées peut être réalisée. Cette comparaison ne peut pourtant mettre directement en évidence les apports d'une technique ou d'une autre *que* si les mécanismes de base sous-jacents, qui ne font pas l'objet de la comparaison, sont identiques. Ceci n'arrive pourtant pas souvent, car cette condition est difficile à remplir.

Une solution possible à ce problème serait la mise en œuvre d'une *plate-forme d'expérimentation* pour des systèmes à mémoire virtuellement partagée, qui intègre les composants de base (comme le support des communications, le mécanisme de protection des pages, etc.). L'utilisateur peut alors mettre en œuvre des nouveaux modèles ou protocoles de cohérence par une implémentation plus rapide et plus facile, en utilisant les *mêmes* mécanismes de base (déjà disponibles) et peut ainsi effectuer des comparaisons concluantes. Cette idée a été exploitée dans une certaine mesure par quelques MVP, mais les choix laissés au programmeur sont souvent limités. Le système TreadMarks [4] a servi de cadre d'implémentation pour plusieurs protocoles de cohérence à la libération en mode paresseux et plusieurs études comparatives ont été publiées. Néanmoins, ces études ont été réalisées par des modifications successives du système par ses concepteurs, car la sélection d'un protocole ou d'un autre n'était pas censée faire partie de l'interface de programmation. Munin [19] laisse plus de liberté au programmeur, en lui permettant de sélectionner lui-même les protocoles implicitement, à travers des annotations, mais les protocoles implémentent unique-

ment la cohérence à la libération en mode immédiat. Ni Munin, ni TreadMarks ne sont pas conçus pour fournir une plate-forme d'expérimentation à des concepteurs de *nouveaux* modèles ou protocoles. Seul le projet CVM [52] a mentionné cet objectif, sans pourtant l'atteindre, car le projet a été arrêté avant que la première version stable soit prête. La conception d'une telle plate-forme est donc restée un objectif important non encore atteint. C'est précisément l'objectif de notre travail.

2.4 Multithreading et mémoire virtuellement partagée

Dans la section précédente nous avons présenté la notion de mémoire virtuellement partagée et nous avons décrit quelques aspects importants de la conception de systèmes à MVP. Nous avons aussi montré pourquoi une plate-forme d'expérimentation de protocoles de cohérence est nécessaire. Pendant notre thèse, nous sommes intéressé à la conception d'une telle plate-forme. Plus particulièrement, notre recherche s'est effectuée dans le cadre de l'environnement multithread PM^2. Notre objectif a donc été de concevoir une plate-forme d'implémentation de protocoles de cohérence *multithread*. Dans cette section nous présentons quelques efforts de recherche dans des directions proches de la nôtre. En particulier, nous analysons les raisons qui ont motivé l'intégration des threads dans les systèmes à MVP et nous donnons un aperçu de quelques systèmes qui ont réalisé cette intégration.

Nous avons présenté dans la section 2.2 les différentes utilisations du multithreading au sein des environnements de programmation parallèle, et tout particulièrement dans le cas des architectures distribuées, exploitées par des systèmes comme PM^2. L'utilisation de multithreading dans les environnements parallèles distribués a été le plus souvent motivée par le gain en efficacité obtenu grâce au parallélisme de tâches implémentées par des processus légers (threads), à l'opposé des implémentations à base de processus lourds. Les threads offrent en plus la possibilité de recouvrir efficacement des communications par des calculs. Enfin, le modèle de programmation multithread connaît une utilisation de plus en plus large, grâce aux différentes implémentations de bibliothèques de threads POSIX, ou à travers le langage Java. Il est alors devenu intéressant que les systèmes MVP supportent le multithreading au niveau de l'interface de programmation d'applications. Plusieurs efforts de recherche ont été réalisés pour fournir des environnements de programmation à MVP supportant le multithreading. En voici quelques exemples.

Millipede. L'un des premiers systèmes à mémoire virtuellement partagée qui intègrent le multithreading est Millipede [33, 48, 47]. Ce système a été développé sous la coordination d'Assaf Schuster (Technion, Israël) avec l'objectif de fournir une machine virtuelle dotée d'une interface permettant la programmation des environnements distribués d'une manière semblable à la programmation des machines multiprocesseurs à mémoire partagée (SMP). Les applications visées sont des programmes multithreads déjà parallélisés en vue d'une exécution sur des machines SMP, qui nécessiteraient très peu de modifications pour une exécution sur la machine virtuelle de Millipede. Les directions de recherche explorées par Millipede sont principalement liées à l'ordonnancement des threads dans un environnement distribué et à l'équilibrage de charge par migration de threads, dans un contexte où la mémoire est virtuellement partagée. Dans Millipede, la migration des threads est guidée par un module d'équilibrage de charge basé sur des historiques d'accès des threads aux données. Par ailleurs, les données migrent aussi : les accès des threads déterminent la migration de ces données (ou leur réplication), suite aux actions du protocole de cohérence. Millipede implémente un seul modèle de cohérence, la cohérence séquentielle, et vise un environnement

d'exécution spécifique, des grappes de PC sous Windows NT dont les nœuds sont reliés par un réseau rapide tel que Myrinet. Un autre thème de recherche abordé récemment par Millipede concerne les problèmes de faux partage dans les MVP à base de pages. La solution proposée, appelée MultiView [47, 82] permet la mise en œuvre d'une granularité de l'ordre de la taille des données individuelles et repose sur un mécanisme d'association de plusieurs "vues" à la même page physique. Chaque vue correspond à une donnée et peut être gérée individuellement.

Brazos. Un autre système à MVP conçu pour s'exécuter dans un environnement Windows NT est Brazos [98]. Ce système développé par John K. Bennet à Rice University, intègre le multithreading à deux niveaux. Au niveau de l'implémentation de la mémoire partagée, les threads sont utilisés pour permettre le recouvrement des communications par des calculs. Au niveau applicatif, les codes utilisateur multithreads sont également supportés. Nous remarquerons que l'implémentation est basée sur un multithreading de *niveau système*, dont la gestion est moins efficace que dans le cas d'un ordonnancement en espace utilisateur. Nous remarquerons aussi que Brazos propose un seul modèle de cohérence, la cohérence de portée et cible une architecture particulière, des serveurs multiprocesseurs Compaq Proliant 1500 sous Windows NT 4.0 reliés par Fast Ethernet.

DSM-Threads. DSM-Threads [75, 76, 77] est le premier système multithread distribué qui implémente le standard POSIX pour les threads via une mémoire virtuellement partagée. L'objectif est similaire à celui de Millipede : il s'agit d'exécuter en environnement distribué des applications multithreads POSIX initialement écrites pour des machines monoprocesseurs ou multiprocesseurs à mémoire partagée. DSM-Threads implémente donc une extension du modèle de programmation défini par POSIX tout en prenant en charge les aspects liés à la distribution. Il propose plusieurs modèles de cohérence : cohérence séquentielle, cohérence à la libération et cohérence d'entrée. Conçu initialement pour des grappes de PC sous SunOS et Linux interconnectées par un réseau Ethernet sous TCP, DSM-Threads a été récemment porté sur une version préliminaire de la bibliothèque de communications de PM^2 (Madeleine). Ce travail vise une meilleure portabilité du système sur l'ensemble de réseaux et d'interfaces de communications supportés par Madeleine et il est actuellement en phase d'optimisation.

CVM. L'environnement CVM [52] (*Coherent Virtual Machine*), développé sous la coordination de Peter Keleher (University of Maryland) est un système à mémoire virtuellement partagée dont l'un des objectifs principaux a été de fournir une plate-forme d'implémentation et de comparaison de protocoles de cohérence, à l'opposé des systèmes mentionnés ci-dessus qui se limitent à fournir à l'utilisateur une MVP fermée, sans possibilité d'évolution. CVM a été développé en C++ et définit des classes génériques permettant l'implémentation de protocoles spécifiques à partir de ces classes. Initialement conçu selon le schéma classique d'un système à MVP monothread, CVM a proposé par la suite d'intégrer le multithreading. Ce projet, dont les objectifs ont été particulièrement proches des nôtres, a été arrêté récemment, avant que la première version multithread stable soit opérationnelle.

DOSMOS. DOSMOS [90] est un environnement de programmation et d'exécution à base de mémoire virtuellement partagée à base d'objets conçu au LIP (ENS Lyon) sous la coordination de Lionel Brunie. Ce système propose un modèle de cohérence hiérarchique dérivé du modèle de cohérence à la libération. Il adresse le problème du faux partage en permettant un découpage des variables en sous-variables dont la cohérence est gérée indépendamment, de manière à minimiser les communications sous-jacentes. Il propose également une struc-

turation des processus en groupes hiérarchiques et des mécanismes de gestion de la cohérence à l'intérieur des groupes, toujours avec l'objectif de minimiser les communications. Un premier prototype à base de processus (lourds) a été implémenté par Laurent Lefevre [59]. Olivier Reymann [90] a proposé une implémentation beaucoup plus efficace, qui utilise de manière interne les threads fournis par le système PM^2. L'utilisation du multithreading est ici motivée uniquement par des considérations d'efficacité de l'implémentation du système. DOSMOS ne fournit pas d'interface de programmation permettant le partage des données par des *threads* de niveau utilisateur, au niveau applicatif.

Des problématiques assez proches de celles posées par l'intégration des threads dans les systèmes à MVP ont été mises en évidence lors de la conception de systèmes à MVP pour grappes de machines multiprocesseurs à mémoire partagée (SMP), tels que Cashmere-2L [99, 27, 55], Shasta [27] ou HLRC-SMP [12, 92]. Dans ces systèmes, plusieurs processeurs au sein d'un nœud partagent la même page physique et la cohérence est assurée grâce à des protocoles à deux niveaux : le premier niveau gère la cohérence inter-nœud et le deuxième gère la cohérence intra-nœud (et utilise généralement le partage physique de la mémoire au sein du nœud). Trois types d'approches sont possibles dans ce contexte.

1. Le premier consiste à faire abstraction du partage physique de la mêmoire au sein d'un nœud et d'associer à chaque processeur une table des pages et d'autres structures de gestion de la cohérence, *sans aucun partage* d'informations entre les processeurs du même nœud.
2. Le deuxième type d'approche est complètement opposé. Il consiste à utiliser une seule table des pages pour tous les processeurs d'un nœud et de faire partager *toutes* les informations sur l'état des pages partagées au sein d'un même nœud. Par conséquent, toute page invalidée par un processeur sera invalidée pour tous les processeurs du nœuds, ce qui est parfois pénalisant. En revanche, toute copie de page transférée par un processeur peut également servir à d'autres processeurs du même nœud, ce qui réduit le nombre de défauts de pages, ainsi que les communications générées pour satisfaire les accès correspondants.
3. Enfin, un troisième type d'approche consiste à utiliser une stratégie hybride et à partager seulement *une partie* des informations sur l'état des pages. Des systèmes comme Cashmere-2L [99] et HLRC-SMP [12, 92] utilisent deux niveaux de structures de données de gestion de la cohérence, ayant pour but de minimiser les communications inter-nœuds tout en gardant des approches paresseuses permettant à chaque processeur d'un nœud de ne prendre en compte que les informations disponibles sur ce nœud qui concernent le processeur concerné. Les processeurs d'un même nœud ont des vues "privées" de l'état des pages partagées et peuvent avoir de droits d'accès différents à une même page.

Bien que les problématiques exposées par les systèmes à MVP pour des grappes de machines multiprocesseurs soient proches de celles spécifiques aux MVP multithreads pour grappes de machines monoprocesseurs, une différence fondamentale les sépare. Dans le premier cas, les processus qui s'exécutent sur un nœud ont des espaces d'adressage virtuel disjoints. Par conséquent, les trois approches mentionnées ci-dessus sont envisageables pour la gestion de la cohérence. Dans le deuxième cas, les threads qui s'exécutent sur un même nœud utilisent le même espace d'adressage virtuel. Par conséquent, la seule approche valable pour une MVP multithread implémentée au niveau logiciel et basée sur le mécanisme de gestion de la mémoire virtuelle consiste à utiliser une seule table des pages pour tous les threads d'un même nœud (de manière similaire à la deuxième approche exposée ci-dessus). En effet, les droits d'accès à la page sont associés au pro-

cessus englobant et s'appliquent pour tous les threads contenus, qui ne peuvent pas gérer d'états différents pour la même page.

L'ensemble des systèmes multithreads à MVP présentés représentent des efforts de recherche importants qui ont mis en évidence des problèmes spécifiques à la conception des systèmes multithreads à mémoire virtuellement partagée. Ces systèmes présentent pourtant un certain nombre de limitations. À l'exception de CVM, ce sont des environnements fermés, qui proposent un ou plusieurs modèles de cohérence et ciblent généralement une architecture spécifique. Généralement, ils offrent très peu de possibilités de configuration permettant la mise en œuvre d'optimisations liées aux applications exécutées. Seul CVM s'est proposé de fournir une plate-forme ouverte, permettant l'ajout de protocoles par l'utilisateur, mais malheureusement cet objectif n'a pas été atteint.

Ces constats constituent le point de départ de notre thèse. Notre principal objectif est de concevoir une plate-forme qui permette l'implémentation et l'évaluation de protocoles de cohérence pour mémoire virtuellement partagée dans un contexte multithread, dans un cadre unifié. Un tel type de système peut être particulièrement utile aux concepteurs de modèles et protocoles de cohérence. Nous nous sommes proposé de construire un système ouvert, extensible et portable, interfaçable avec des environnements de programmation parallèle de haut niveau, auxquels il peut offrir un support d'implémentation. En effet, un tel système à MVP peut constituer la cible d'un système de compilation de langage basé sur le paradigme de la mémoire partagée. L'efficacité d'une telle approche rend nécessaire une autre qualité du système, sa flexibilité. L'interface de programmation du système doit fournir l'accès à des mécanismes internes (éventuellement alternatifs), permettant des expérimentations basées sur différentes solutions.

2.5 Conclusion

Dans ce chapitre, nous avons présenté la notion de *multithreading* et nous avons discuté son utilisation pour l'exploitation efficace des architectures distribuées, dans des environnements de programmation tels que PM^2. Nous avons ensuite montré que la programmation de ce type d'architecture nécessite un effort de programmation important et que, à ce propos, un progrès important peut être réalisé grâce à la *transparence* de la localisation des données et des traitements. La notion de *mémoire virtuellement partagée* introduit cette transparence vis-à-vis de l'utilisateur : les threads peuvent accéder des données partagées indépendamment de la localisation des threads et des données. Nous avons introduit les concepts de base liés à la notion MVP et les principales problématiques associées. Nous avons ensuite brièvement décrit quelques systèmes à MVP qui ont intégré le multithreading. Enfin, nous avons mis en évidence le besoin d'une plate-forme d'implémentation et d'expérimentation de protocoles de cohérence multithread pour mémoire virtuellement partagée.

Le travail présenté dans cette thèse est centré sur la conception et l'implémentation d'une telle plate-forme. Cette plate-forme, que nous avons appelé DSM-PM^2 (*Distributed Shared Memory - Parallel Multithreaded Machine*), a été construite à partir des mécanismes de base de l'environnement multithread PM^2. Les principes de conception, les détails d'implémentation et les applications de la plate-forme font l'objet des chapitres suivants.

Chapitre 3

Notre proposition : la plate-forme DSM-PM^2

Ce chapitre présente la plate-forme DSM-PM^2. Notre objectif a été de proposer un cadre unifié qui facilite l'implémentation, l'expérimentation comparative et l'évaluation de protocoles de cohérence pour mémoire virtuellement partagée, dans un contexte multithread. Nous avons commencé par identifier les événements qui déclenchent habituellement des actions de cohérence dans les MVP existantes et nous avons défini la notion de *protocole de cohérence* comme étant un ensemble de routines associées à ces événements. Cette approche est détaillée dans la section 3.1, qui présente également une liste de mécanismes génériques, indépendants des protocoles, qui constituent une couche de base au-dessus de laquelle le concepteur peut facilement implémenter différents protocoles. L'architecture de DSM-PM^2 est ensuite présentée dans la section 3.2 et son principe de fonctionnement est donné dans la section 3.3. La section 3.4 introduit les protocoles de cohérence fournis par la bibliothèque de DSM-PM^2 et met en évidence quelques aspects liés à leur implémentation multithread. Enfin, la section 3.5 montre comment le programmeur peut utiliser ces protocoles de bibliothèque et comment il peut en définir de nouveaux.

3.1 Conception d'une MVP générique

3.1.1 Cadre d'étude

Notre étude vise à généraliser les approches illustrées par des systèmes tels que DSM-Threads [75, 76] et Millipede [48], deux systèmes multithreads à MVP. Notre objectif n'est pas simplement de proposer une MVP multithread de plus, mais de fournir une plate-forme *ouverte*, *configurable*, permettant la comparaison de protocoles alternatifs pour le même modèle de cohérence. Plus encore, nous proposons une plate-forme *extensible*, sur laquelle on puisse facilement implémenter de *nouveaux* protocoles de cohérence grâce à un support *générique*. Cette approche est illustrée par la figure 3.1.

L'interface de programmation de DSM-PM^2 a été conçue avec un double objectif. Premièrement, elle permet des expérimentations rapides à l'aide des protocoles intégrés dans la plate-forme ou des protocoles fournis par le programmeur, pour des modèles de cohérence classiques. Nous avons pris en compte le modèle de *cohérence séquentielle* et les modèles de cohérence faibles tels

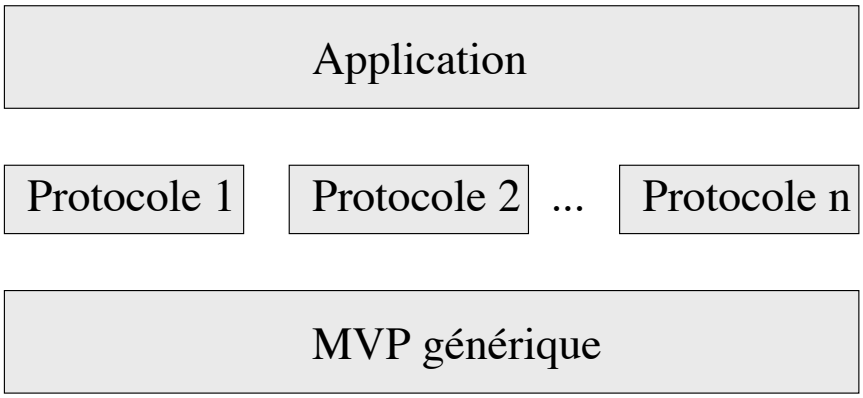


FIG. 3.1 – Schéma d'utilisation d'une MVP générique.

que la *cohérence à la libération*. Nous pensons que le support générique à son état actuel permet également l'implémentation de protocoles pour la *cohérence d'entrée* et pour la *cohérence de portée*.

Deuxièmement, la plate-forme DSM-PM^2 a été conçue pour servir de cible pour des systèmes de compilation de langages basés sur la mémoire partagée qui nécessitent l'implémentation de protocoles de cohérence spécifiques. Nous avons illustré ce type d'utilisation par l'interfaçage de DSM-PM^2 avec Hyperion, un système de compilation Java pour grappes de PC, pour lequel nous avons conçu et évalué deux protocoles spécifiques, qui implémentent la *cohérence Java*. Dans l'avenir, nous allons également nous intéresser à la compilation du langage OpenMP.

La notion de *MVP générique* a été dégagée en étudiant les mécanismes mis en œuvre dans les MVP *à base de pages*, implémentées entièrement au niveau *logiciel*, en espace *utilisateur*. Nos motivations pour ce choix ont été multiples. Tout d'abord, ces MVP sont largement majoritaires et elles ont illustré un large spectre de modèles et protocoles de cohérence. Ensuite, une implémentation en espace utilisateur, sans support spécifique du système d'exploitation, assure la portabilité de notre plate-forme et convient parfaitement à une utilisation expérimentale sur plusieurs architectures différentes. Enfin, nous supposons qu'aucune analyse de la localité des données n'est faite en amont. La détection de la localité est à la charge de DSM-PM^2 et utilise le mécanisme matériel de protection des pages virtuelles pour associer des actions de cohérence aux défauts de pages. La granularité utilisée est donc celle de la page. Dans le cas où DSM-PM^2 est utilisé comme cible d'un compilateur et que les accès aux données partagées sont contrôlés par des primitives spécifiques, l'utilisation des défauts de page pour la détection de localité n'est plus indispensable, car l'implémentation de ces primitives permet d'effectuer des *tests* de localité. Nous avons illustré ces deux approches lors de l'implémentation de nos deux protocoles de cohérence Java.

DSM-PM^2 est disponible sur plusieurs systèmes de type Unix (Linux 2.2, Solaris 2.7, AIX 4.1). Nos expérimentations ont été faites sur des grappes de taille moyenne (16 nœuds), utilisant différents types de réseaux (Fast Ethernet, Myrinet, SCI).

3.1.2 Définition d'un protocole de cohérence

Une étude détaillée des protocoles utilisées par les MVP à base de pages permet d'identifier une liste restreinte d'événements qui déclenchent des actions de cohérence. Une première liste

Fonction	Description
routine de gestion des défauts de page en lecture	Appelée lors d'un défaut de page en lecture
routine de gestion des défauts de page en écriture	Appelée lors d'un défaut de page en écriture
serveur des requêtes en lecture	Appelé lors de la réception d'une requête de page en lecture
serveur des requêtes en écriture	Appelé lors de la réception d'une requête de page en écriture
serveur des requêtes d'invalidation	Appelé lors de la réception d'une requête d'invalidation de page
serveur de réception des pages	Appelé lors de la réception d'une page
routine *acquire*	Appelée lors du blocage d'un verrou
routine *release*	Appelée lors de la libération un verrou
routine d'initialisation	Appelée lors de l'association d'une page au protocole

TAB. 3.1 – Fonctions qui définissent un protocole dans DSM-PM^2.

peut être déduite implicitement des présentations données par Li et Hudak [64] pour leurs protocoles de cohérence séquentielle. Il s'agit essentiellement des défauts de page (en lecture ou en écriture), des réceptions de requêtes de page (également en lecture ou en écriture) et des réceptions d'invalidations de page. Les protocoles de Li et Hudak sont en effet définis à travers 5 fonctions, correspondant aux 5 événements que nous venons de mentionner. En nous inspirant de cette approche, nous pouvons concevoir notre MVP générique en introduisant la possibilité d'associer des *actions de cohérence* spécifiques à un protocole à un certain nombre fixe d'*événements génériques*. Tout nouveau protocole de cohérence peut alors être spécifié en donnant la liste des fonctions à exécuter pour chaque événement générique.

Notre objectif étant de concevoir une MVP supportant plusieurs modèles de cohérence, nous devons généraliser la liste précédente d'événements de manière à prendre en compte les modèles qui ont succédé à la cohérence séquentielle illustrée par les protocoles de Li et Hudak. Ainsi, pour le cas des protocoles de cohérence faible, différentes actions sont exécutées lors des opérations de synchronisation (*acquire* et *release*). D'autre part, les opérations à effectuer lors de la *réception des pages* sur un nœud peuvent varier d'un protocole à un autre. Enfin, certaines actions initiales peuvent être nécessaires à l'*initialisation*, selon les protocoles, par exemple pour protéger certains types d'accès aux pages contrôlées par le protocole.

Dans la version actuelle de DSM-PM^2, nous avons défini 9 événements. Leur liste détaillée est donnée dans la table 3.1. Définir un protocole de cohérence dans DSM-PM^2 consiste à fournir un ensemble de 9 routines, une pour chaque événement mentionné dans la table 3.1, destinées à être appelées automatiquement par le support générique. Ces 9 événements génériques ont été définis suite à une analyse d'un large spectre de protocoles existants. Néanmoins, une extension du support générique à plus d'événements est facilement envisageable dans l'avenir si les progrès dans le domaine des protocoles de cohérence mettent en évidence un tel besoin.

Les 9 routines peuvent être définies en utilisant l'interface de programmation des composants de base de DSM-PM^2 (la table des pages, les routines de communication). Suivant le protocole,

certaines de ces fonctions peuvent être non définies. Selon notre expérience personnelle, la complexité du code correspondant aux routines reste maîtrisable : il s'agit de quelques dizaines de lignes par routine.

3.1.3 Mécanismes de base pour une MVP générique

Afin de faciliter l'implémentation de protocoles de cohérence, nous avons identifié un certain nombre de mécanismes communs à la plupart des systèmes MVP existants à base de pages, pouvant servir de support d'implémentation. L'ensemble de ces mécanismes constituent ce que nous appelons *MVP générique*. Ils fournissent une couche de base facilitant l'implémentation des protocoles de cohérence.

Gestion des pages. Les MVP à base de pages utilisent une table qui stocke des informations sur les pages partagées. Cette table peut être centralisée ou distribuée sur les différents nœuds. Chaque page est gérée individuellement par les protocoles, en fonction des informations qui lui correspondent. Certaines informations sont utilisées par la plupart des protocoles : les droits d'accès locaux à la page, le propriétaire courant, etc. D'autres peuvent être spécifiques à certains protocoles. Une MVP générique doit fournir une telle table des pages, pourvue de fonctions permettant de manipuler des informations associées aux pages. En plus, la structure de cette table doit être conçue de manière à permettre l'ajout de nouveaux champs d'information si nécessaire.

Gestion des communications. Nous pouvons constater que les protocoles de cohérence illustrés par les MVP existantes utilisent un ensemble limité de routines de communication. Les opérations les plus habituelles sont la transmission des requêtes de pages, la transmission des pages, les messages d'invalidation des pages et les messages d'acquittement associés. Les protocoles à écrivains multiples ont également besoin de transmettre non pas des pages entières, mais des modifications effectuées localement sur les pages. Une MVP générique doit également fournir un tel ensemble de routines qui puisse être utilisé par plusieurs protocoles.

Synchronisation et cohérence. Les modèles de cohérence faible, tels que la cohérence à la libération, la cohérence d'entrée ou la cohérence de portée, exigent que des actions de cohérence soient effectuées lors des opérations de synchronisation. Afin de supporter de tels modèles, les MVP fournissent des objets de synchronisation (verrous, barrières) et des primitives associées. Une MVP générique doit proposer de tels objets et fournir un mécanisme permettant d'associer des actions de cohérence aux opérations de synchronisation sur ces objets.

Détection d'accès. Afin d'assurer la cohérence des données partagées, les MVP ont besoin d'un mécanisme de détection des accès à ces données. Certains systèmes, qui intègrent un compilateur, utilisent à cet effet des notations au niveau du langage, ou bien introduisent des primitives spécifiques d'accès aux données. La plupart des systèmes évitent pourtant l'utilisation des compilateurs et l'accès aux données partagées se fait par simple affectation ou lecture. Le mécanisme de protection des pages de la mémoire virtuelle est alors utilisé par la MVP pour déterminer l'état des données et éventuellement déclencher des actions de cohérence. En règle générale, si la donnée est présente en mémoire locale et que sa valeur est à jour, l'accès est réduit au coût d'une affectation. Sinon, l'accès déclenche un défaut de page, suite auquel la MVP effectue les opérations nécessaires pour mettre à jour la mémoire locale. Certains protocoles utilisent les défauts de page même si la mémoire est à jour, par exemple

pour identifier les pages modifiées localement et permettre la transmission des modifications aux autres nœuds plus tard. Dans tous les cas, les défauts de page déclenchent des actions destinées à assurer la cohérence, qui peuvent varier d'un protocole à un autre. Une MVP générique doit donc fournir le support pour permettre de détecter les défauts de page, d'extraire des informations sur ces défauts de page (type : lecture ou écriture, adresse, etc.) et d'y associer des actions spécifiques aux protocoles.

Une MVP générique *multithread*

Notre étude étant effectuée dans le cadre d'un environnement multithread, nous nous sommes particulièrement intéressé aux aspects spécifiques à ce contexte. Le multithreading a été intégré récemment dans les MVP, afin d'en améliorer l'efficacité, mais aussi pour supporter le modèle de programmation associé aux threads, devenu de plus en plus utilisé. À la différence des MVP classiques, une MVP multithread doit prendre en compte l'exécution *concurrente* d'actions de cohérence sur la même page ou sur des pages différentes. Cela a des conséquences à plusieurs niveaux.

- D'une part, les structures de données faisant partie du support générique, telles que la table des pages, doivent prévoir des mécanismes de synchronisation pour la gestion de multiples accès concurrents aux informations concernant la même page.
- Par ailleurs, l'utilisation de ces mécanismes de base dans les protocoles de cohérence est également plus complexe, car les protocoles doivent utiliser des routines réentrantes, capables d'être exécutées de manière concurrente par plusieurs threads de l'application qui accèdent aux données partagées.
- Enfin, de manière transparente pour le développeur de protocoles, le multithreading peut servir pour l'implémentation de la MVP générique elle-même. Plusieurs threads de service peuvent, par exemple, être créés pour différents services indépendants.

3.2 Architecture de DSM-PM^2

La distinction entre mécanismes génériques et actions spécifiques aux protocoles a pour conséquence immédiate la structuration en couches de DSM-PM^2 (figure 3.2). Au niveau supérieur, la *bibliothèque de protocoles* permet de construire des protocoles de cohérence à partir des *briques de base* fournies par la couche inférieure. Ces briques de bases sont des routines utilisables par plusieurs protocoles, construites au-dessus des principaux composants génériques de DSM-PM^2 : le *gestionnaire des pages* et le *gestionnaire des communications*.

3.2.1 Le gestionnaire des pages

Le *gestionnaire des pages* maintient une table des pages partagées et gère les droits d'accès aux pages sur les différents nœuds. Pour chaque page partagée allouée, la table conserve un certain nombre d'informations, telles que le numéro du nœud propriétaire, les droits d'accès sur le nœud local, éventuellement un ensemble de copies, etc. Le concepteur de protocole peut utiliser cette table suivant le type de gestion souhaité (gestionnaire centralisé ou distribué, fixe ou dynamique, cf. section 2.3.2), car cette gestion est spécifiée au niveau du protocole. En effet, ce sont les routines

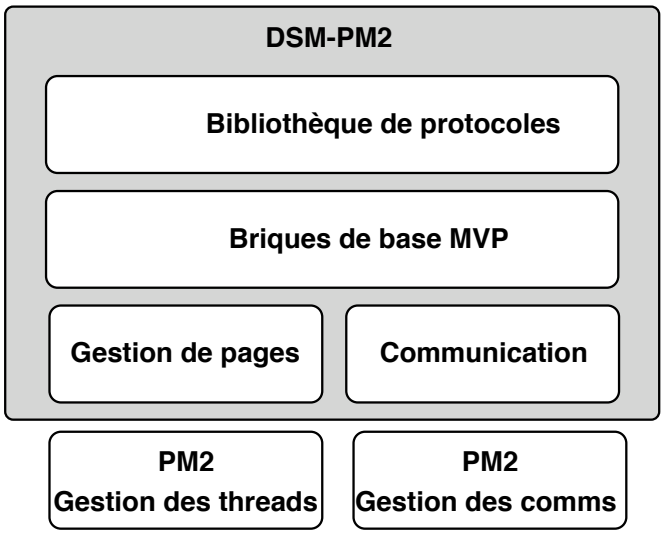


FIG. 3.2 – L'architecture logicielle de DSM-PM^2.

spécifiques au protocole qui manipulent les entrées de la table et qui donnent une interprétation aux informations qui y sont stockées. La table est suffisamment générique pour permettre des gestions assez différentes. La bibliothèque interne de protocoles de DSM-PM^2 comporte, par exemple, des protocoles à gestionnaire distribué fixe et des protocoles à gestionnaire distribué dynamique. Cette polyvalence de la structure de la table des pages a pour conséquence le fait qu'un champ présent dans une entrée de la table peut avoir des sémantiques différentes dans des protocoles différents, alors qu'il peut rester inutilisé par certains protocoles. Par exemple, dans un protocole basé sur une gestion fixe de la propriété des pages, le champ *owner* de chaque entrée de la table indique le propriétaire courant de la page correspondante. Dans le cas d'une gestion dynamique, le même champ peut être utilisé pour indiquer le propriétaire *probable* de la page.

De plus, la table est extensible : de nouveaux champs peuvent être facilement ajoutés si de nouveaux protocoles l'exigent. Trois champs de type entier sont disponibles, ainsi qu'un champ de type `void*`, permettant de rattacher toute autre structure de données à chaque entrée de la table. S'il utilise ces champs, le concepteur de protocole doit également enregistrer une fonction d'initialisation, que DSM-PM^2 appellera automatiquement lors de l'allocation de toute nouvelle entrée dans la table.

Un aspect important de la conception de cette table est le fait qu'elle est destinée à être utilisée dans un contexte multithread. Des primitives sont fournies pour permettre la synchronisation des threads qui accèdent à la même entrée de la table de manière concurrente, à l'aide des sections critiques (`dsm_lock_page`, `dsm_unlock_page`) ou par des mécanismes d'attente et de signalisation (`dsm_wait_for_page`, `dsm_signal_page_ready`). Les principales routines de synchronisation sont décrites dans la table 3.2.

3.2.2 Le gestionnaire des communications

Le *gestionnaire des communications* fournit des routines de base utilisées dans la plupart des MVP existantes : envoi des requêtes de pages, envoi des pages, envoi des messages d'invalidation.

Fonction	Rôle
`dsm_lock_page`	Verrouiller l'entrée de la table des pages pour une page donnée (entrée en section critique).
`dsm_unlock_page`	Déverrouiller l'entrée de la table des pages pour une page donnée (sortie d'une section critique).
`dsm_wait_for_page`	Attendre l'actualisation de l'entrée de la table des pages pour une page donnée. Le thread appelant est bloqué est sera réveillé dès qu'un autre thread appelle `dsm_signal_page_ready` pour la même page.
`dsm_signal_page_ready`	Signaler l'actualisation de l'entrée de la table des pages pour une page donnée. Réveille tous les threads bloqués par les appels à `dsm_wait_for_page` pour la même page.

TAB. 3.2 – Principales routines de synchronisation fournies par le gestionnaire des pages de DSM-PM^2.

Il comporte également des routines spécifiques à certains classes de protocoles : envoi des *différences*, utiles pour les protocoles à écrivains multiples, ou envois groupés de plusieurs requêtes ou pages en un message unique. Les principales routines proposées sont énumérées dans la table 3.3. D'autres routines, non listées ici, sont disponibles pour permettre des envois groupés en un seul message pour *plusieurs* pages, requêtes de pages, ou différences vers le même nœud.

Toutes les routines de communication de DSM-PM^2 sont implémentées en utilisant le mécanisme des RPC de PM^2, qui s'avère particulièrement approprié à cette tâche. Par exemple, envoyer une requête de page revient à invoquer un service à distance sur le nœud qui détient la page. Les services sont généralement asynchrones, à l'exception de l'envoi des *différences*, qui nécessite un acquittement confirmant leur prise en compte sur le nœud destination. Ce type de service synchrone est implémenté en utilisant les RPC de manière conjointe avec un mécanisme de sémaphores distribués fourni par PM^2. Deux solutions sont proposées par DSM-PM^2 : l'envoi

Fonction	Rôle
`dsm_send_page_req`	Envoyer une requête de page.
`dsm_send_page`	Envoyer une page.
`dsm_send_invalidate_req`	Envoyer une requête d'invalidation de page.
`dsm_send_invalidate_ack`	Envoyer un acquittement confirmant l'invalidation d'une page.
`dsm_send_diffs`	Envoyer les différences associées à une page et attendre l'acquittement.
`dsm_send_diffs_start`	Envoyer les différences associées à une page.
`dsm_send_diffs_wait`	Attendre l'acquittement des différences.

TAB. 3.3 – Principales routines du gestionnaire des communications de DSM-PM^2.

synchrone, avec attente immédiate de l'acquittement, et l'envoi avec attente différée, qui permet au nœud ayant envoyé les différences d'effectuer d'autres traitements avant d'attendre l'acquittement. Notre approche utilise donc les RPC et suppose, de manière générale, qu'un mécanisme de ce type est fourni par les couches logicielles sous-jacentes.

L'utilisation d'une interface de programmation de haut niveau telle que celle des RPC présente un deuxième avantage : la portabilité. En effet, le mécanisme des RPC de PM^2 est basé sur MADELEINE, la bibliothèque de communications de PM^2, et hérite ainsi la portabilité de cette bibliothèque sur un grand nombre d'interfaces de communication : BIP, SISCI, MPI, VIA, TCP, etc. Par conséquent, la plate-forme DSM-PM^2, dont les communications sont implémentées en termes de RPC, est également portable sur un grand nombre de types de réseaux. Cette portabilité est extrêmement utile dans notre contexte, car elle permet la réalisation d'études comparatives de performances sur des architectures ayant des caractéristiques différentes. La flexibilité de DSM-PM^2, en tant que plate-forme expérimentale, est ainsi renforcée.

3.2.3 Les briques de base

Les protocoles intégrés fournis par DSM-PM^2 se composent de fonctions construites en utilisant l'interface de programmation des deux composants de base que nous venons de décrire, le gestionnaire des pages et le gestionnaire des communications. Ces routines (*briques de base*) sont définies dans la couche intermédiaire de DSM-PM^2 et certaines sont partagées par plusieurs protocoles intégrés. De manière générale, ces fonctions sont destinées à faciliter l'implémentation de nouveaux protocoles de cohérence, dans lesquels elles pourraient être éventuellement réutilisées.

Une caractéristique commune de ces routines est leur conception adaptée à un environnement multithread. Si, dans les MVP classiques, on peut supposer qu'une telle routine s'exécute de manière atomique, cette hypothèse n'est plus valable dans un contexte multithread. Par conséquent, ces routines doivent prendre en compte leur éventuelle exécution concurrente. Par exemple, plusieurs défauts de page concernant la *même* page peuvent se produire de manière concurrente, alors que l'on souhaite qu'une seule requête de page soit envoyée au propriétaire. Ou encore, les droits d'accès locaux à une page peuvent changer entre le moment où un défaut de page se produit et le moment où l'on s'apprête à envoyer une requête. Dans la section 4.1 du chapitre suivant nous décrirons de manière plus détaillée les situations de concurrence les plus communes et les solutions adoptées dans les protocoles intégrés.

3.2.4 La bibliothèque de protocoles

La couche la plus haute de DSM-PM^2 est la *bibliothèque de protocoles*. Cette couche comporte essentiellement une table, dans laquelle chaque entrée spécifie les 9 fonctions qui correspondent à un protocole donné. À l'initialisation, la table comprend aujourd'hui 6 protocoles intégrés, qui implémentent 3 modèles de cohérence : 2 pour la cohérence séquentielle, 2 pour la cohérence à la libération et 2 pour la cohérence Java. Les caractéristiques de ces protocoles seront présentées dans la section 3.4. Elles sont résumées dans la table 3.4.

Notre objectif étant de fournir une plate-forme qui facilite les études expérimentales sur des protocoles, et tout particulièrement sur de *nouveaux* protocoles non intégrés à ce jour dans la plate-forme, une propriété nécessaire de notre table des protocoles est son extensibilité. Aussi, l'interface de programmation de cette couche de haut niveau permet de rajouter de nouveaux protocoles

dans la table. Cet ajout est fait à l'aide d'une primitive, `dsm_create_protocol`, qui permet de définir de nouveaux protocoles à partir de 9 routines, à spécifier. Ces routines peuvent faire partie de la couche des briques de base, mais peuvent aussi être fournies par le programmeur. La section 3.5 illustre l'utilisation de cette primitive.

3.2.5 Mécanismes de synchronisation

Les modèles de cohérence faible associent des actions de cohérence aux opérations de synchronisation. DSM-PM^2 fournit deux types d'objets de synchronisation : les verrous et les barrières. Nous rappelons que le blocage d'un verrou correspond à une action *acquire*, alors qu'une libération de verrou correspond à un *release*. Enfin, une barrière correspond à un *release* suivi d'un *acquire* [36]. En effet, si deux processus P_1 et P_2 se synchronisent par barrière afin que P_2 puisse lire la valeur d'une variable x écrite par P_1, il est nécessaire que P_1 transmette la valeur de x *avant* la synchronisation proprement dite (c.-à-d. avant que P_2 confirme à P_1 son arrivée à la barrière). Dans les modèles de cohérence faible, cette opération est effectuée lors du *release*. De même, P_2 doit mettre à jour sa mémoire locale, afin de pouvoir lire la valeur à jour de x, *après* que P_1 ait transmis cette valeur. Cette mise à jour est effectuée lors du *acquire*.

Dans DSM-PM^2, nous avons choisi de permettre au programmeur d'associer explicitement un protocole à un verrou. C'est grâce à cette association que le blocage du verrou déclenche l'action *acquire* du protocole et que la libération du verrou déclenche l'action *release*. Le fonctionnement des barrières est similaire.

Les verrous. L'interface de programmation de DSM-PM^2 fournit le type `dsm_lock_t`, que le programmeur peut utiliser pour déclarer un verrou global, utilisé pour protéger des données stockées dans la MVP. Un thread peut bloquer un verrou en utilisant la primitive `dsm_lock` et le libérer en appelant `dsm_unlock`. Avant toute utilisation, le verrou doit être initialisé à l'aide de la primitive `dsm_lock_init`, qui permet d'associer un ou plusieurs protocoles de cohérence au verrou. Cette association informe indirectement DSM-PM^2 sur les actions de cohérence à effectuer à l'entrée, respectivement à la sortie des sections critiques protégées par le verrou. En effet, la primitive `dsm_lock` appelle implicitement les routines *acquire* de tous protocoles associés au verrou, juste après le blocage du verrou. De même, la primitive `dsm_unlock` appelle toutes les routines *release* des protocoles associés, juste avant la libération du verrou.

Les barrières. DSM-PM^2 fournit également des objets de type barrière : `dsm_thread_barrier_t`, pour des barrières auxquelles peuvent participer un nombre arbitraire de threads de l'application (éventuellement supérieur au nombre de nœuds, sous condition de participation d'au moins un thread par nœud). La synchronisation par barrière se fait par la primitive `dsm_thread_barrier`. Son implémentation repose sur une synchronisation locale des threads de chaque nœud suivie d'une barrière au niveau des nœuds. Étant donné que l'arrivée à une barrière correspond à un *release* et que le départ d'une barrière correspond à un *acquire*, l'opération de synchronisation fait appel à ces deux routines du protocole associé à la barrière. Comme dans le cas des verrous, cette association est effectuée lors de l'initialisation des objets barrière.

3.3 Principe de fonctionnement de DSM-PM^2

3.3.1 Principe général

Le principe de base de DSM-PM^2 consiste à capter les événements génériques énumérés dans la section 3.1.2 et à exécuter pour chacun de ces événements la routine correspondante du protocole de cohérence. Les "points d'entrée" dans un protocole sont généralement les défauts de page et les opérations de synchronisation.

Défauts de page. Lorsqu'un thread accède à une donnée partagée absente de la mémoire locale, cet accès génère habituellement un défaut de page. DSM-PM^2 capte le défaut de page, détermine l'identité de la page accédée et déroute le thread vers la *routine de gestion du défaut de page en lecture* ou vers la *routine de gestion du défaut de page en écriture*. Dans beaucoup de protocoles tels que les protocoles MRSW ou les protocoles MRMW à référence, cette routine consulte la table des pages pour déterminer le propriétaire ou le nœud de référence de la page accédée et lui demande la page. Le thread se bloque ensuite, en attente de la page.

Il faut remarquer que l'interface du système d'exploitation ne fournit pas toujours d'information sur le type d'accès qui a produit le défaut de page (lecture ou écriture). Si cette information manque et que le nœud a déjà les droits d'accès en lecture, il s'agit d'un défaut de page en écriture. Si le nœud n'a aucun droit d'accès, c'est la routine de gestion des défauts de page en *lecture* qui est appelée par défaut. Une fois que le nœud obtient la page en lecture, l'accès est répété et un deuxième défaut de page se produit éventuellement s'il s'agit d'un accès en écriture. Dans ce cas, les droits d'accès en lecture étant déjà acquis, DSM-PM^2 appelle la routine de gestion des défauts de page en *écriture*.

Réception d'une requête de page. La réception d'une requête de page sur un nœud déclenche l'exécution d'une autre routine du protocole : le *serveur de requêtes en lecture* ou le *serveur de requêtes en écriture*, suivant le type de requête. Dans le cas des protocoles à gestionnaire fixe, c'est toujours le propriétaire de la page qui reçoit la requête. Dans le cas des protocoles à gestionnaire dynamique, chaque nœud connaît uniquement le propriétaire *probable* de la page et, par conséquent, la requête peut arriver sur un nœud non propriétaire. Les serveurs de requêtes de ce type de protocole commencent habituellement par tester si le nœud local est réellement propriétaire. Si ce n'est pas le cas, la requête est transmise au propriétaire probable tel que spécifié par le gestionnaire local des pages.

Généralement, le traitement exécuté par les serveur de requêtes sur le nœud propriétaire consiste à envoyer la page au nœud demandeur, après quelques actions préliminaires qui peuvent varier d'un protocole à un autre. Par exemple, dans les protocoles MRSW, le propriétaire doit d'abord invalider toutes les copies de la page, en envoyant des requêtes d'invalidation aux nœuds concernés. La page est envoyée dès que ces invalidations sont confirmées par des acquittements.

Réception d'une requête d'invalidation. Lorsqu'un nœud reçoit une requête d'invalidation, il exécutera le *serveur des requêtes d'invalidation* spécifié par le protocole. Généralement, cette routine invalide la copie locale de la page et renvoie un acquittement au nœud ayant demandé l'invalidation. Dans certains protocoles, l'ensemble des copies d'une page est distribué sur plusieurs nœuds sous la forme d'un arbre dont la racine est le propriétaire de la page. Dans ce cas, le *serveur des requêtes d'invalidation* envoie typiquement à son tour des requêtes d'invalidation à d'autres nœuds, attend les acquittements correspondants et renvoie ensuite son propre acquittement.

Réception d'une page. La réception d'une page par un nœud déclenche l'exécution du *serveur de réception des pages* spécifié par le protocole. Généralement, cette routine positionne les droits d'accès à la page sur le nœud local et réveille le ou les threads bloqués en attente de la page.

Opérations de synchronisation. Comme nous l'avons expliqué dans la section 3.2.5, le blocage d'un verrou, la libération d'un verrou, ainsi que les barrières déclenchent implicitement l'exécution des routines *acquire* et/ou *release* spécifiées par le protocole associé à l'objet de synchronisation concerné.

Initialisation. Enfin, l'initialisation du gestionnaire des pages au lancement de l'application par DSM-PM^2, ainsi que toute allocation dynamique de mémoire partagée fait appel à la routine d'*initialisation* du protocole associé à la donnée. Cette routine positionne généralement les droits d'accès initiaux à la page, tels qu'ils sont spécifiés par le protocole.

Dans la figure 3.3, nous illustrons l'enchaînement des différentes routines intervenant suite à un défaut de page en écriture, dans le cas d'un protocole MRSW. Nous supposons que le défaut de page a lieu sur le nœud 0, que le nœud 1 est propriétaire de la page et que le nœud 2 en possède une copie en lecture. La propriété de la page est transférée du nœud 1 au nœud 0, après invalidation de la copie du nœud 2. Dans cet exemple, 5 routines du protocole sont invoquées avant que le thread puisse accéder la page.

3.3.2 Support générique et spécificités des protocoles

Lorsqu'un thread accède à des données partagées et génère un défaut de page qui déclenche le protocole de cohérence, DSM-PM^2 exécute deux types de traitements : des traitements *génériques*, exécutés quel que soit le protocole de cohérence, et des traitements *spécifiques* au protocole. Dans la figure 3.4, nous illustrons l'enchaînement de ces traitements dans le cas d'un défaut de page en lecture.

Lors d'un défaut de page, le support générique de DSM-PM^2 prend la main et consulte une table, pour déterminer s'il s'agit d'un accès à une donnée partagée ou bien si c'est simplement un défaut de segment ordinaire (*segmentation fault*) dû à une erreur de programmation. S'il s'agit d'une donnée partagée, le gestionnaire des pages partagées est interrogé afin de déterminer le protocole de cohérence associé à la page. Cette information est stockée pour chaque page dans l'entrée correspondante de la table des pages partagées sous la forme d'un identifiant de type entier. Cet identifiant correspond à la position du protocole recherché dans la table des protocoles. Chaque élément de cette table contient les adresses des 9 routines qui définissent un protocole. Une fois que l'entrée correspondant au protocole est trouvée, la *routine de gestion des défauts de page en lecture* spécifiée dans la table est appelée. C'est à ce moment que le support générique passe la main au protocole, lui permettant d'exécuter des traitements spécifiques.

Cette implémentation a deux avantages importants. Tout d'abord, elle permet d'associer un protocole à chaque page, et par conséquent des pages différentes peuvent être gérées par des protocoles divers. DSM-PM^2 est donc une plate-forme multiprotocole qui permet non seulement d'exécuter la même application avec des protocoles différents, mais qui permet également la coexistence de plusieurs protocoles pour des données différentes au sein de la même exécution. Deuxièmement, il faut remarquer que le protocole est déterminé à l'exécution, par une consultation de la table des pages partagées. Cela permet la mise en œuvre de stratégies basées sur un changement dynamique du protocole associé à une page, en fonction du comportement de l'application. Une telle modification demande néanmoins un effort de la part du programmeur : il doit

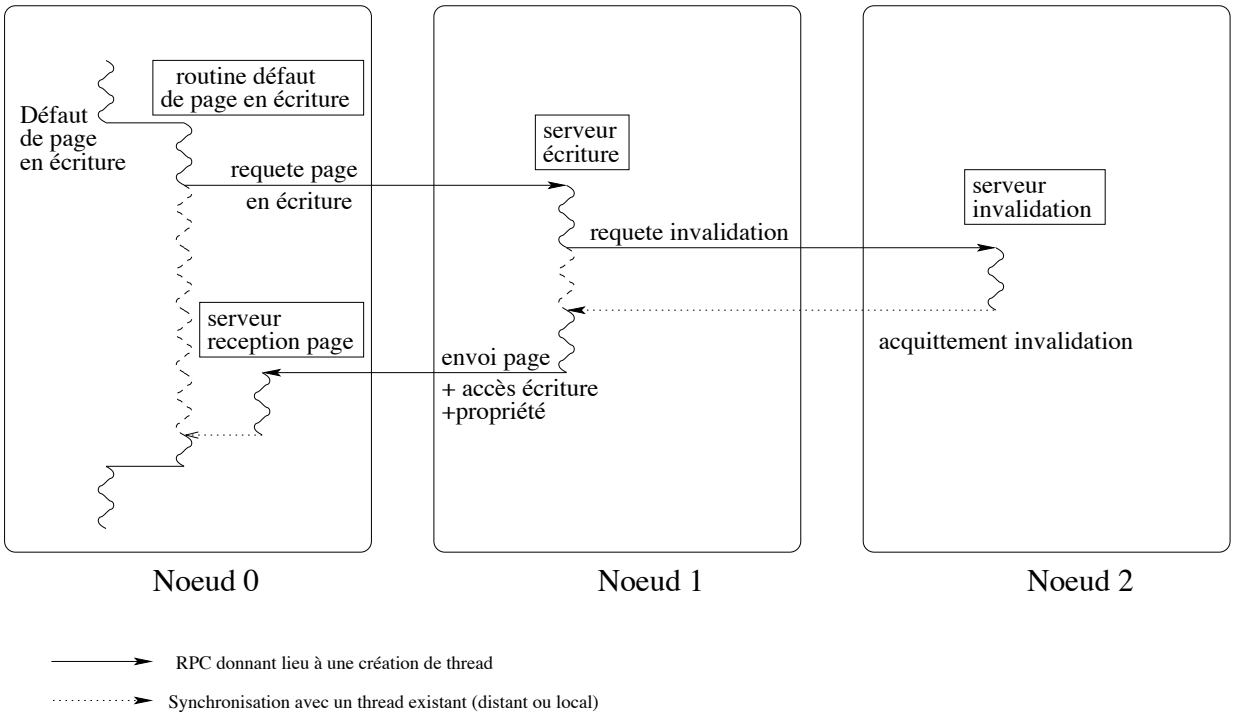


FIG. 3.3 – Exécution d'un protocole de cohérence MRSW lors d'un défaut de page en écriture.

Lorsqu'un thread T qui s'exécute sur le nœud 0 génère un défaut de page en écriture, il est dérouté par DSM-PM^2 vers la routine qui gère cet événement dans le protocole associé à la page. Dans cet exemple, la routine demande la page au propriétaire (le nœud 1). Cette requête déclenche l'exécution du "serveur écriture" du protocole sur le nœud 1. Avant d'envoyer la page au nœud 0, la routine serveur demande l'invalidation de la copie de la page détenue par le nœud 2, qui exécute à cet effet le "serveur invalidation". Une fois cette invalidation confirmée, le nœud 1 envoie la page au nœud 0, avec les droits d'écriture et le droit de propriété de la page. La réception de la page est effectuée par le "serveur réception page", qui réveille le thread T une fois la page installée. Nous remarquerons que tous les services sont exécutés par des threads créés dynamiquement pour le traitement des requêtes. Par contre, la routine de gestion du défaut de page est exécutée par le thread ayant déclenché le défaut.

utiliser des synchronisations globales pour s'assurer que l'initialisation du nouveau protocole sur tous les nœuds est effectuée en l'absence de tout accès à la page concernée.

3.4 Modèles de cohérence et protocoles multithreads dans DSM-PM^2

Dans la version courante, DSM-PM^2 fournit 6 protocoles de cohérence intégrés, dont les principales caractéristiques sont résumées dans la table 3.4. Trois modèles de cohérence sont supportés : la cohérence séquentielle, la cohérence à la libération et la cohérence Java. Afin d'illustrer la richesse des mécanismes de base fournis, nous avons construit 2 protocoles de cohérence pour chacun de ces modèles, en utilisant des mécanismes alternatifs.

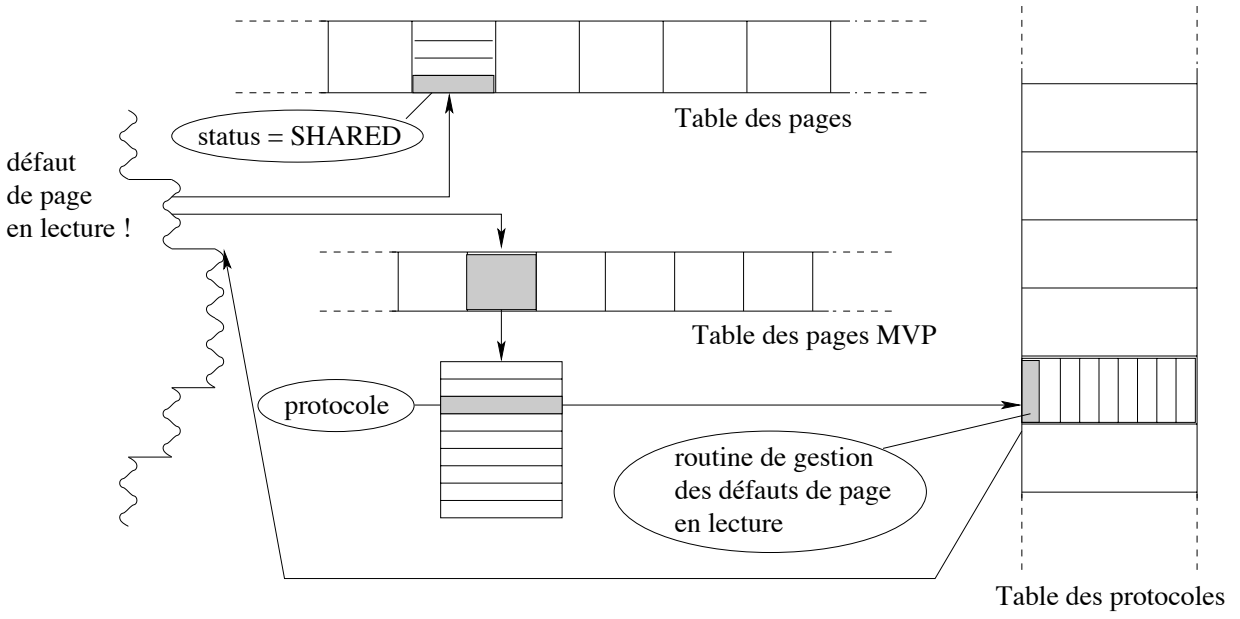


FIG. 3.4 – Actions génériques et actions spécifiques au protocole lors d'un défaut de page en lecture.

Cette section présente quelques aspects qui doivent être pris en compte lors de la conception de protocoles de cohérence multithread et décrit ensuite les 6 protocoles multithreads intégrés fournis par DSM-PM^2. Nous avons choisi de présenter l'ensemble de ces protocoles afin d'illustrer comment chacun des trois modèles de cohérence peut être implémenté dans DSM-PM^2 par des mécanismes différents. Nous présentons le principe de fonctionnement, sans détailler, dans un premier temps, les aspects liés à la concurrence. Nous énumérons ensuite les problèmes supplémentaires posés par l'exécution concurrente de plusieurs threads par nœud. Nous invitons le lecteur à se référer au chapitre 4 pour une présentation plus détaillée de ces aspects.

3.4.1 Conception de protocoles de cohérence multithreads

Les protocoles proposés par DSM-PM^2 partagent une importante caractéristique commune : ils sont fonctionnels dans un contexte *multithread*. Par conséquent, ils sont souvent plus complexes que les protocoles classiques dont ils s'inspirent, à cause de la multiplicité des situations de concurrence à gérer. Deux threads peuvent déclencher des défauts de page concurrents sur la même page, ou bien une requête de page peut s'exécuter en concurrence avec une demande d'invalidation de la page. L'implémentation doit alors garantir le respect des contraintes de cohérence tout en permettant un degré élevé de concurrence. Ceci permet une exploitation plus efficace des processeurs disponibles (en particulier grâce aux recouvrements des temps d'attente ou de communication par des calculs), mais l'implémentation est particulièrement délicate, car la combinatoire des actions est difficile à maîtriser.

Exécution concurrente de la *même* routine d'un protocole. Une première conséquence est que les routines qui composent les protocole de cohérence doivent pouvoir être exécutées de

manière concurrente par plusieurs threads du même nœud, et même lorsqu'il s'agit de la même page. Par exemple, lorsqu'un thread déclenche un défaut de page en lecture, il exécute la routine correspondante du protocole, qui typiquement envoie une requête au propriétaire de la page. Si un deuxième thread déclenche un autre défaut de page en lecture sur la même page juste après l'envoi de la requête, il n'a pas besoin d'envoyer une deuxième requête, mais doit simplement se bloquer en attente de la page. Enfin, si un troisième thread accède en lecture la même page et que la page arrive entre le moment où le thread déclenche son défaut de page et le moment où il exécute la routine de gestion du défaut, le thread peut simplement retourner de cette routine, sans attendre. Nous retrouvons les mêmes situations pour des défauts de page concurrents en écriture.

Dans les protocoles monothreads classiques, cette réentrance n'est pas exigée : tout défaut de page déclenche systématiquement une requête, car il n'y a pas d'autre thread qui ait pu envoyer une requête précédemment. Ceci est dû à l'exigence sémantique d'*atomicité* de la routine de gestion du défaut de page : aucun défaut de page n'est traité avant que le traitement du précédent défaut concernant la même page soit fini. Dans un contexte multithread, une éventuelle sérialisation des requêtes reste possible, mais réduit fortement le gain potentiel en performance qui serait obtenu en permettant l'attente d'un thread sur une page par le traitement d'un autre thread prêt à s'exécuter.

Exécution concurrente des *différentes* routines d'un protocole. Plus encore, les MVP monothread classiques font l'hypothèse que *toutes* les routines des protocoles sont atomiques. Dans un contexte multithread, cette hypothèse est trop contraignante. Si un thread est bloqué en attente d'une page suite à un défaut de page en lecture et qu'un autre thread déclenche un défaut de page en écriture sur la même page, il n'y a aucune raison d'attendre la réception de la page en lecture avant d'envoyer une requête en écriture : les deux routines de gestion des deux types de défaut de pages doivent pouvoir s'exécuter de manière concurrente. Ce type de situation se retrouve souvent dans les protocoles multithreads et nécessite parfois la mise en œuvre de stratégies de gestion particulièrement complexes.

Exécution concurrente des actions de cohérence sur des *pages différentes*. Enfin, lorsqu'il s'agit de pages différentes, la concurrence peut être totale, car les données de gestion utilisées par DSM-PM^2 sont disjointes dans ce cas. Toute routine de tout protocole peut généralement s'exécuter de manière concurrente à toute autre routine du même protocole, ou d'un autre protocole, si cette deuxième routine concerne une page différente. Seuls les protocoles qui utilisent des structures de données auxiliaires (par exemple, des listes de pages) peuvent parfois avoir besoin de protéger temporairement les accès à ces structures par des sections critiques. Le reste du temps, toute concurrence est possible et cela permet de tirer profit des recouvrements possibles dans un environnement multithread tel que PM^2.

Discussion méthodologique

La correction des protocoles de cohérence classiques proposés dans la littérature repose souvent sur l'exigence d'*atomicité* des traitements des défauts de page. Les protocoles traditionnels implémente cette atomicité en sérialisant les traitement correspondants. Une solution triviale fréquemment utilisée consiste à protéger les traitements des routines par des verrous acquis à l'entrée et libérés à la sortie des routines. Ce type d'approche peut également être implémenté dans un contexte multithread. Pourtant, si la solution est correcte, l'efficacité qu'elle engendre est particulièrement réduite.

Comme nous l'avons montré dans les exemples cités plus haut, l'implémentation des protocoles de cohérence multithreads peut tirer profit de plusieurs techniques disponibles dans un environnement multithread : le recouvrement potentiels des temps d'attente ou de communication d'un thread bloqué par des calculs exécutés par un thread, ou bien la factorisation de certains traitements. La mise en œuvre de ces techniques nécessite pourtant une implémentation plus complexe des protocoles. L'enjeu est d'une part de réduire la portée des sections critiques et de permettre un degré de concurrence élevée et d'autre part de mettre en œuvre des mécanismes de coopération et de synchronisation entre les différentes routines, afin de leur permettre de factoriser correctement leurs traitements. Ce sont précisément ces stratégies que nous avons adoptées lors de la conception de nos protocoles de cohérence multithreads.

Notre approche pose deux problèmes importants. Le premier concerne la correction des protocoles obtenus par rapport aux exigences sémantiques des modèles de cohérence alors que les traitements des routines du protocole ne sont plus atomiques. Le deuxième est lié à la quantification du gain obtenu grâce aux optimisations mises en œuvre. Les contraintes temporelles imposées par le déroulement de cette thèse ne nous ont pas permis d'aborder ces deux aspects de manière satisfaisante.

- Le problème de la correction a déjà été abordé par Frank Mueller pour son implémentation d'une version multithread d'un protocole de Li et Hudak [75, 76], par la proposition d'une modélisation à base d'automates. Cette modélisation qui permet de démontrer la correction d'un protocole pourrait être étendue aux protocoles que nous avons proposés.
- La quantification des gains en efficacité peut être réalisée en deux étapes. D'une part, une instrumentation des protocoles est nécessaire afin de déterminer la fréquence des situations de concurrence abordées. D'autre part, il est important de mesurer les gains en temps d'exécution obtenus sur des applications réalistes lorsque ces optimisation sont activées par rapport à une implémentation triviale qui sérialise les traitements.

Même si ces deux aspects n'ont pas été traités de manière complète durant notre thèse, les protocoles que nous proposons ont été testés soigneusement sur des applications réelles, comme nous le montrons dans le chapitre suivant. Nos observations expérimentales confirment que les situations de concurrence abordées se produisent fréquemment et que les gains obtenus grâce à nos optimisations sont significatifs. Une quantification exacte reste à réaliser.

3.4.2 Cohérence séquentielle

La bibliothèque de DSM-PM^2 fournit 2 protocoles qui implémentent la cohérence séquentielle en utilisant des mécanismes différents. Ainsi, le premier est basé sur la migration des pages, alors que le deuxième sur la migration des threads. Le premier admet la coexistence de multiples copies de la même page en lecture seule, ce qui n'est pas le cas du deuxième protocole, qui n'utilise pas de réplication.

Le protocole `li_hudak`

Le protocole `li_hudak` repose sur l'algorithme MRSW introduit par Li et Hudak [64], basé sur un gestionnaire distribué dynamique. Cet algorithme a été adapté à une exécution multithread par Mueller [75]. Nous avons utilisé une variante de cette version multithread, dont les caractéristiques particulières sont présentées ci-dessous.

Protocole	Modèle de cohérence	Principales caractéristiques
`li_hudak`	séquentielle	Protocole MRSW. Réplication des pages en lecture, changement de propriétaire lors des accès en écriture. Gestionnaire distribué dynamique.
`migrate_thread`	séquentielle	Protocole SRSW. Utilise sur la migration de threads vers les données lorsque les accès déclenchent des défauts de page. Gestionnaire distribué fixe.
`erc_sw` (*eager release consistency, single writer*)	libération	Protocole MRSW. Implémente la cohérence à la libération en mode *immédiat*. Gestionnaire distribué dynamique.
`hbrc_mw` (*home-based release consistency, multiple writers*)	libération	Protocole MRMW. Implémente la cohérence à la libération. Stratégie *immédiate*, *à référence* (*home-based*). Utilise des copies jumelles. Différences calculées au *release*, envoyées immédiatement au nœud de référence. Gestionnaire distribué fixe.
`java_ic` (*Java consistency, inline checks*)	Java	Protocole MRMW. Stratégie *à référence* (*home-based*). Utilise des *tests explicites* pour déterminer la localité. Différences calculées et enregistrées à la volée, lors des écritures. Gestionnaire distribué fixe.
`java_pf` (*Java consistency, page faults*)	Java	Protocole MRMW. Stratégie *à référence* (*home-based*). Utilise des *défauts de page* pour déterminer la localité. Différences calculées et enregistrées à la volée, lors des écritures. Gestionnaire distribué fixe.

TAB. 3.4 – Protocoles intégrés dans la bibliothèque de DSM-PM^2.

Principe. L'algorithme utilise la réplication en lecture et la migration à l'écriture. Lors d'un défaut de page en lecture, la routine de gestion de cet événement demande une copie de la page au nœud propriétaire de cette page tel qu'il est indiqué par la table locale des pages partagées. Le thread appelant se bloque ensuite en attente de la page. Lors d'un défaut de page en écriture, le traitement est similaire, sauf dans le cas où le défaut de page a lieu sur le nœud propriétaire même ; ceci peut arriver lorsque la page est répliquée en lecture, situation dans laquelle toutes les copies, y compris celle du nœud propriétaire, sont en lecture seule. Dans ce cas, aucune requête de page n'est envoyée : le nœud propriétaire invalide les copies et ensuite autorise la modification de sa copie.

Lors de la réception d'une requête de page en lecture sur un nœud, le *serveur de requêtes de pages en lecture* vérifie si le nœud local possède une copie de la page. Si c'est le cas, une copie de cette page est envoyée au demandeur. Sinon, la requête est transmise au propriétaire probable, tel qu'il est indiqué par la table locale. Remarquons que tout nœud possédant une copie de la page peut répondre à la requête (et non seulement le propriétaire).

Dans le cas d'une requête de page en écriture, celle-ci est transmise jusqu'au propriétaire, qui invalide toutes les copies éventuelles de la page avant de demander la migration de la page vers le nœud demandeur, qui obtient ainsi le droit de modifier la page et en devient le propriétaire (cf. figure 3.3).

Enfin, l'arrivée de la page sur le nœud demandeur déclenche l'exécution du *serveur de réception des pages*, qui positionne les droits d'accès correspondants et réveille le(s) thread(s) en attente.

Gestionnaire distribué dynamique. Dans ce protocole, chaque nœud connaît le propriétaire *probable* de chaque page. À l'initialisation, cette information est à jour sur tous les nœuds. Lorsqu'une page change de propriétaire, c'est uniquement les tables des deux nœuds source et destination qui sont mises à jour. Un nœud tiers qui accède la page en écriture par la suite adresse une requête à l'ancien propriétaire, qui la fera suivre au nouveau propriétaire, qui répond en envoyant la page. Par la même occasion, les tables de ces trois nœuds sont actualisées, car c'est le troisième nœud qui devient maintenant propriétaire de la page.

De même, lors de la réception d'une page en lecture, le *serveur de réception des pages* met à jour le propriétaire probable dans la table du nœud récepteur : il s'agit du nœud ayant répondu à la requête. Un certain nombre de nœuds obtiendront cette copie du vrai propriétaire et pourront la transmettre ensuite à d'autres nœuds. L'ensemble des copies est donc organisé sous la forme d'un arbre orienté, dont la racine est le nœud propriétaire et dont les arêtes sont définies par la relation *propriétaire probable*.

Li et Hudak montrent [64] que cet algorithme permet de toujours retrouver le propriétaire d'une page, le nombre nécessaire de messages étant borné par le nombre de nœuds.

Distribution de l'ensemble des copies. L'ensemble des copies étant organisé sous la forme d'un arbre, chaque nœud garde un sous-ensemble de l'ensemble des copies, correspondant aux nœuds auxquels il a transmis une copie en lecture. Tout accès en écriture déclenche l'invalidation de toutes les copies, à commencer par celle du propriétaire courant. Lorsqu'un nœud reçoit une invalidation, il la propage vers les nœuds du niveau inférieur (s'il y en a), attend les acquittements correspondants et renvoie son acquittement au nœud qui lui a envoyé la requête d'invalidation. Lorsque le propriétaire courant a reçu tous les acquittements, la page peut être migrée vers le nouveau propriétaire.

Aspects spécifiques au contexte multithread. Le protocole permet un degré relativement élevé de concurrence : des défauts de page peuvent être traités en parallèle même lorsqu'ils

concernent la même page. Comme nous l'avons expliqué dans la section 3.4.1, la requête envoyée suite à un défaut de page peut servir plusieurs défauts de page quasi-simultanés du même type, sur la même page. Si un défaut de page en écriture arrive pendant l'exécution de la routine de gestion des défauts de page en lecture, il sera traité de manière concurrente et engendrera une deuxième requête, sans attendre que la requête en lecture soit satisfaite. Les routines de gestion des défauts de page peuvent même s'exécuter en parallèle avec les autres routines du protocole. Par exemple, un nœud peut demander une copie en lecture d'une page à un autre nœud en attente du droit d'écrire sur la page. Sur ce deuxième nœud, un thread est bloqué dans la *routine de gestion du défaut de page en écriture*, mais cela n'empêche pas l'exécution du *serveur des requêtes en lecture*.

Le protocole `migrate_thread`

DSM-PM^2 propose aussi un autre protocole qui implémente le même modèle de cohérence séquentielle en utilisant la migration de threads.

Principe Le principe de ce protocole est très simple et il est illustré sur la figure 3.5. Le protocole n'utilise ni la réplication, ni la migration des pages : une page n'est accessible que sur son nœud propriétaire. Lors d'un défaut de page, qu'il soit en lecture ou en écriture, la routine de gestion du défaut de page consulte la table des pages locale pour déterminer le nœud propriétaire de la page et ensuite migre le thread vers ce nœud. Dans ce protocole, la propriété des pages ne change jamais, donc la requête est toujours envoyée au vrai propriétaire. Après migration, le thread répète l'accès à la page (avec succès) et ensuite continue son exécution sur ce nœud. Le protocole repose essentiellement sur une seule fonction, utilisée en tant que routine de gestion des défauts de page, à la fois en lecture et en écriture, et qui ne fait que migrer le thread sur le nœud propriétaire de la page. *Les 7 autres fonctions du protocoles sont vides !*

En contrepartie de cette simplicité, les accès concurrents à la même donnée sur des nœuds différents ne sont pas possibles (il s'agit d'un protocole SRSW). Par conséquent, tous les threads qui accèdent la même page migrent sur le nœud propriétaire de cette page. Même si la migration d'un thread est généralement bien moins coûteuse que celle d'une page à cause de la taille plus réduite des données à transférer, l'efficacité globale de ce protocole dépend de manière significative de la distribution des données partagées sur les différents nœuds, qui a un impact important sur l'équilibre de la charge. Une manière d'aborder ce problème pourrait consister à utiliser des modules d'équilibrage dynamique de la charge par migration des threads. L'étude de l'interaction d'un tel module avec le protocole de cohérence, basé lui aussi sur la migration de threads, pourrait faire l'objet d'un travail en prolongement de cette thèse.

Le protocole que nous venons de présenter dépend fortement de *l'approche iso-adresse* que nous avons utilisée pour l'allocation des données partagées : ces données sont toujours placées à la même adresse virtuelle sur tous les noeuds. En sortant de la routine de gestion des défauts de page, après migration, le thread répète automatiquement l'accès à la même adresse virtuelle, qui correspond en effet à la même donnée sur le nœud destination. Nous détaillerons dans le chapitre 4 les extensions que nous avons rajoutées à l'allocateur `isomalloc` de PM^2 pour gérer l'allocation dynamique des données partagées.

Aspects spécifiques au contexte multithread. De par la nature du mécanisme de migration qu'il utilise, ce protocole a été conçu spécifiquement pour un environnement multithread et n'a

pas d'équivalent dans les MVP classiques monothread.

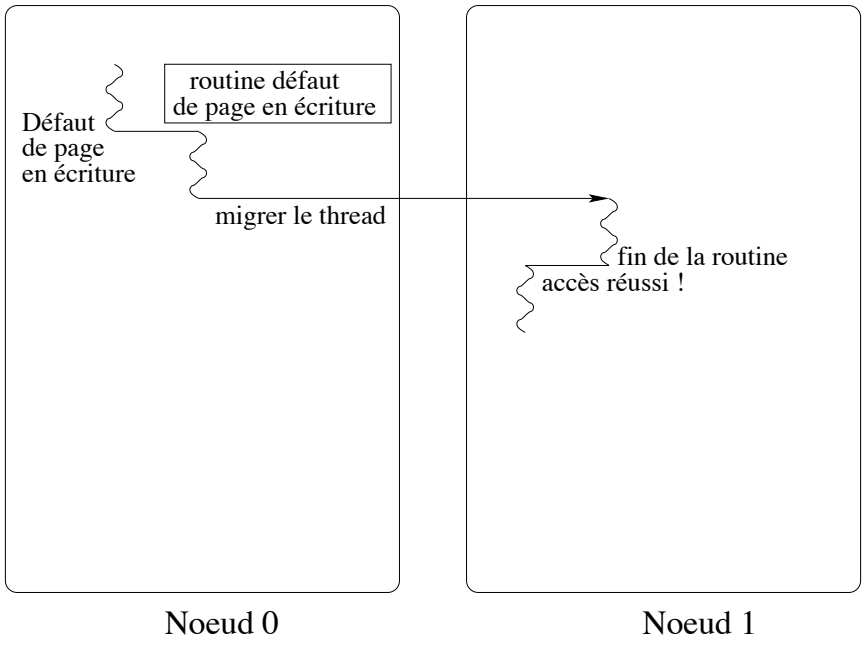


FIG. 3.5 – Cohérence séquentielle implémentée à l'aide de la migration des threads : lors d'un défaut de page, le thread migre vers le nœud propriétaire de la page.

3.4.3 Cohérence à la libération

La bibliothèque de DSM-PM^2 fournit également 2 protocoles qui implémentent la cohérence à la libération. Le premier (`erc_sw`) utilise une stratégie *immédiate*, à écrivain unique. Le deuxième (`hbrc_mw`) est basé sur une stratégie *à référence* et utilise des écrivains multiples. Comme dans le cas du protocole `li_hudak`, le schéma général de ces protocoles est classique [53]. Notre contribution est d'en avoir fourni une version adaptée à un environnement d'exécution multithread, significativement plus complexe à cause des situations de concurrence.

Le protocole `erc_sw`

L'implémentation la plus simple du modèle de cohérence à la libération est basée sur une stratégie *immédiate* : c'est l'opération *release* qui rend visibles les modifications effectuées par un nœud aux données partagées. Le protocole `erc_sw` (*eager release consistency, single writer*) implémente une telle stratégie.

Principe. La spécificité d'un protocole de cohérence à la libération consiste à reporter certaines actions de cohérence consécutives aux modifications des données partagées jusqu'à la prochaine opération *release*, dans le cas d'une stratégie immédiate, ou même jusqu'à l'opération *acquire* suivante, dans le cas d'une stratégie paresseuse. Dans le protocole `erc_sw`, aucune action n'est effectuée lors d'un *acquire*. La cohérence est assurée par les actions effectuées

lors du *release*. Le principe est simple : la gestion de chaque page partagée est confiée à un nœud *propriétaire* qui seul peut y écrire (protocole MRSW). Les autres nœuds peuvent lui demander une copie de la page en lecture. Lorsque le nœud propriétaire d'une page y accède en écriture, il positionne un drapeau. Lors de la sortie de la section critique, le nœud propriétaire invalide toutes les copies en lecture existantes de la page si le drapeau est validé. Ainsi, un autre nœud souhaitant plus tard lire une variable située sur la page devra demander une nouvelle copie en lecture : il lira donc la nouvelle valeur de la variable en question. Cette situation est illustrée sur la figure 3.6. Si un nœud veut écrire sur la page, il doit demander à son propriétaire de lui céder son droit : il en devient alors le nouveau propriétaire. Comme le protocole `li_hudak`, ce protocole utilise un gestionnaire distribué dynamique et un ensemble de copies organisé en arbre.

Aspects spécifiques au contexte multithread. Ce protocole partage toutes les propriétés du protocole `li_hudak` concernant le traitement concurrent des défauts de page, ainsi que l'exécution des routines de gestion des défauts de page de manière concurrente avec les autres routines du protocole. La routine *release* utilise temporairement des sections critiques internes pour invalider l'ensemble des copies d'une page modifiée localement depuis le dernier *acquire*, mais il s'agit de *plusieurs* sections critiques, une par page concernée. Chaque section critique bloque partiellement l'exécution de certaines parties des autres routines du protocole, lorsqu'elles ont besoin d'accéder aux données de gestion concernant la page courante. Dans les autres cas, la routine *release* peut s'exécuter librement en concurrence avec les autres routines du protocole.

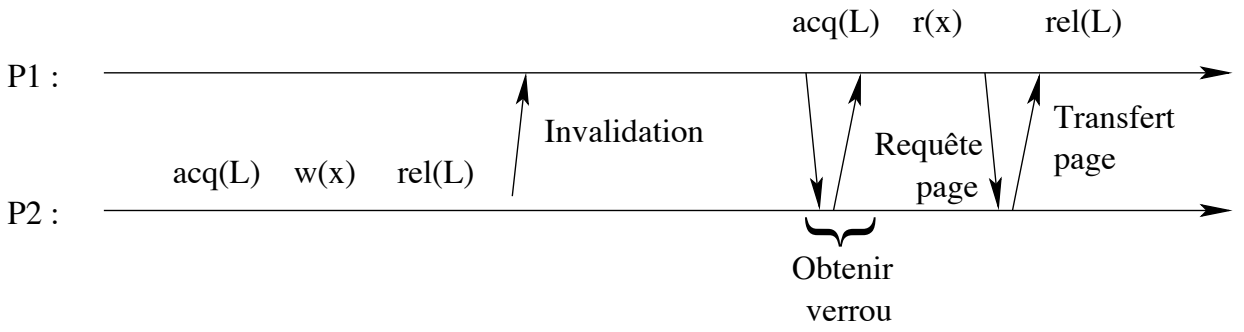


FIG. 3.6 – Le protocole `erc_sw`.

Ce protocole repose sur des invalidations envoyées lors de la sortie des sections critiques. Dans cet exemple, deux processus P_1 et P_2, situés sur des nœuds différents, possèdent initialement une copie de la page contenant la variable x, mais c'est P_2 qui est propriétaire de la page. Le processus P_2 bloque le verrou L, écrit la variable x et ensuite libère le verrou. Lors de cette libération, le processus P_2 invalide la copie de la page détenue par P_1. Supposons que P_1 bloque à son tour le verrou, après P_2, et qu'il lise ensuite la variable x. Une copie de la page est alors redemandée au processus P_2. Observons qu'entre le moment où P_2 écrit la variable x et le moment où P_2 libère le verrou L, la copie modifiable de la page possédée par P_2 coexiste avec la copie en lecture seule de P_1. Pourtant, P_1 verra les modifications de P_2 seulement après que ce dernier ait libéré le verrou, conformément au modèle de cohérence à la libération.

Le protocole `hbrc_mw`

Principe. Le protocole `hbrc_mw` (*home-based release consistency, multiple writers*) est un protocole MRMW *à référence* : chaque page est *statiquement* attachée à un nœud de référence tout au long de l'exécution, ce nœud étant chargé de garder la copie de référence de la page. Il utilise une technique de création de pages jumelles *twinning* [53], une méthode de calcul de différences pour déterminer les parties d'une page qui doivent être mises à jour.

Comme dans le protocole précédent, aucune action n'est effectuée lors de l'entrée d'un thread en section critique. Lorsqu'un thread demande à accéder en lecture à une page partagée, une copie en lecture est ramenée depuis le nœud de référence de la page. Lorsqu'un thread demande un accès en écriture à une page partagée, une copie en lecture est ramenée si nécessaire, puis une sauvegarde *jumelle* (*twin*) en est faite. Le thread peut alors modifier la page. Lorsqu'il quitte la section critique, chaque page modifiée est comparée mot à mot avec sa jumelle. (On suppose par défaut une granularité de 32 bits, comme dans le système TreadMarks [4], qui a été l'un des premiers à populariser cette technique.) L'ensemble des différences détectées est alors envoyé au nœud de référence, qui l'applique immédiatement sur sa copie de la page, après avoir invalidé toutes les autres copies éventuelles. Le principe du protocole est illustré sur la figure 3.7

Ce protocole autorise la plusieurs nœuds à écrire sur la même page en même temps (à des adresses différentes), évitant ainsi les effets de *va-et-vient* induits par le protocole `erc_sw` dans le cas de faux partage. La technique de *twinning* évite de transférer systématiquement des pages entières sur le réseau, mais seulement les différences : ceci représente une économie importante de bande passante réseau. Par contre, elle consomme plus de ressources processeur et mémoire que le précédent : création de la jumelle et comparaison mot à mot de l'original avec la copie. Le compromis entre les deux aspects dépend donc crucialement des schémas d'accès de l'application et des coûts relatifs des mécanismes mis en jeu : coût des transferts *vs* coût de gestion des jumelles.

Aspects spécifiques au contexte multithread. De même que les protocoles `li_hudak` et `erc_sw`, ce protocole autorise le traitement concurrent des défauts de page, ainsi que l'exécution des routines de gestion des défauts de page de manière concurrente avec les autres routines du protocole. Comme dans le cas du protocole `erc_sw`, la routine *release* peut s'exécuter en concurrence avec les autres routines, à l'exception de certaines sections. Une situation de concurrence spécifique à ce protocole peut arriver lorsque deux threads exécutent la routine *release* sur des nœuds différents N_1 et N_2. Si les deux threads ont modifié la même page, les deux vont envoyer les différences vers le nœud de référence de la page. Ce nœud réagit à la réception des premières différences reçues (par exemple du nœud N_1) par une requête d'invalidation adressée au nœud N_2. Cette requête déclenche un envoi des différences qui serait concurrent à l'envoi consécutif au *release* local. Le protocole garantit que les différences sont transmises une seule fois dans cette situation.

3.4.4 Cohérence Java

L'un des objectifs de DSM-PM^2 est de fournir un environnement cible pour des compilateurs de langages utilisant un modèle de programmation à mémoire partagée. Une fois compilés, ces programmes pourront s'exécuter de manière transparente sur des architectures distribuée, telles

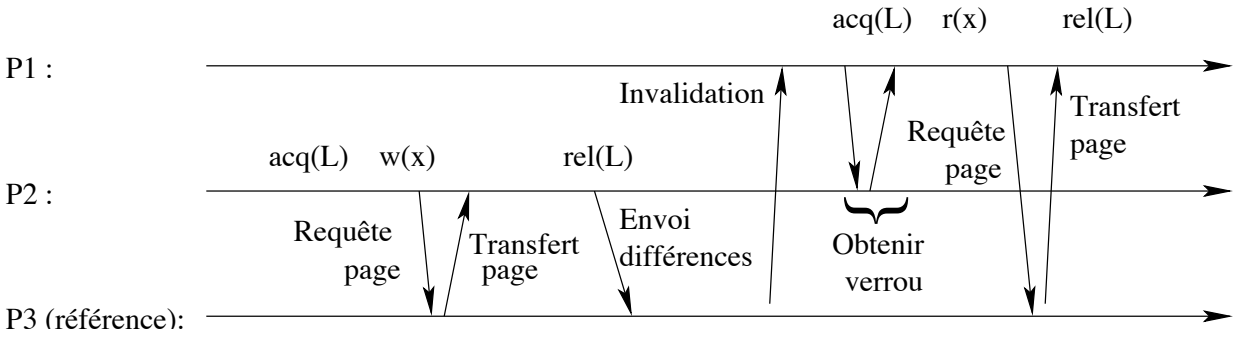


FIG. 3.7 – Le protocole `hbrc_mw`.

Ce protocole utilise la notion de nœud de référence. Tout défaut de page, en lecture ou en écriture, déclenche le transfert de cette page depuis ce nœud. Toute modification apportée à une donnée dans une section critique est transmise au nœud de référence à la sortie de la section. Dans cet exemple, nous supposons encore que deux processus P_1 et P_2, situés sur des nœuds différents, possèdent une copie de la même page, qui contient la variable x. Le processus P_2 bloque le verrou L, écrit la variable x et relâche le verrou. Ce relâchement est effectif après que la nouvelle valeur de x ait été transmise au nœud de référence et que la page correspondante ait été invalidée par P_1. Le processus P_1 bloque ensuite le verrou et relit la variable x. La lecture déclenche le transfert de la page contenant x depuis le nœud de référence.

que les grappes de PC. Nous avons illustré cette approche en intégrant DSM-PM^2 dans un système de compilation Java.

DSM-PM^2 propose deux protocoles qui implémentent directement le modèle de cohérence spécifié par le modèle de mémoire de Java (*Java Memory Model* [38]), que nous désignerons par le terme *cohérence Java*. Ce modèle (très proche de la cohérence à la libération) est présenté de manière détaillée dans le chapitre 5. Grâce à ces protocoles, DSM-PM^2 est utilisé par Hyperion, un système de compilation et d'exécution de programmes Java. Hyperion permet l'exécution *transparente* de programmes multithreads Java sur des grappes de PC. Les deux protocoles ont été conçus de manière conjointe avec le module de gestion de la mémoire d'Hyperion, ce qui nous a permis de mettre en œuvre d'importantes optimisations. Par exemple, un certain nombre d'opérations de synchronisation à l'aide de verrous ont pu être évitées grâce aux garanties fournies par les couches supérieures d'Hyperion.

À la différence des protocoles `li_hudak`, `erc_sw` et `hbrc_mw`, dérivés de spécifications pour des versions monothread, ces deux protocoles implémentent une cohérence spécifiquement conçue pour une exécution multithread. Le modèle de mémoire de Java spécifie la cohérence en termes de *mémoire principale* et *caches des threads*, ce qui n'est pas le cas des autres modèles de cohérence, qui sont spécifiés en termes de nœuds et non pas de threads. L'implémentation de ces deux protocoles dans DSM-PM^2 a été faite en recherchant un degré de concurrence élevé et présente les mêmes propriétés que les autres protocoles quant à la réentrance des routines de gestion des défauts de page.

Dans la section 5.3.2 du chapitre 5 nous discutons de manière détaillée les caractéristiques de ces protocoles. Nous avons effectué une étude comparative de leur comportement à l'aide de plusieurs applications multithreads. Nos tests ont permis d'illustrer la supériorité du protocole

basé sur des défauts de page dans tous les cas étudiés.

3.5 Interface de programmation : spécification des protocoles

L'interface de programmation de DSM-PM^2 permet de déclarer des données partagées *statiques* et *dynamiques* et d'y associer des protocoles de cohérence fournis par la bibliothèque de la plate-forme ou définis par l'utilisateur. Ceci est illustré à travers les exemples présentés dans cette section. Nous commençons par un exemple simple de programme PM^2 (figure 3.8) qui n'utilise pas de partage de données. Les deux exemples suivants illustrent la déclaration et l'utilisation de variables partagées statiques, en utilisant nos deux protocoles de cohérence forte, `li_hudak` (figure 3.9) et `migrate_thread` (figure 3.10). Nous montrons ensuite comment la mémoire partagée est allouée dynamiquement (figure 3.11). L'exemple qui suit présente la sélection dynamique de protocole (figure 3.12) lors de l'exécution du programme. L'utilisation de plusieurs protocoles au sein de la même application est illustrée dans l'exemple de la figure 3.13. Le programme présenté en figure 3.14 introduit l'utilisation des verrous distribués et des protocoles de cohérence faible. Enfin, la définition d'un protocole de cohérence ad-hoc est présentée dans la figure 3.15.

3.5.1 Exécution d'un programme PM^2 sans partage de données

Sur la figure 3.8 nous présentons un programme PM^2 typique écrit pour s'exécuter sur 2 nœuds. Chaque nœud exécute la fonction `pm2_main`, de manière SPMD (*single program, multiple data*). Ici, les deux nœuds appellent la primitive d'initialisation de la barrière `pm2_thread_barrier_init`, ainsi que la primitive d'initialisation générale de PM^2 (`pm2_init`). La primitive `pm2_self` permet à chaque nœud de connaître son identité (ici : 0 ou 1). En testant la valeur retournée par cette primitive, on peut faire exécuter à chaque nœud des instructions différentes. Cet exemple utilise une variable globale `x`, dont chaque nœud possède une copie. Le nœud 0 crée 2 threads, T_0 et T_1, qui exécuteront les fonctions `f0`, respectivement `f1`. Le thread T_1 commence par migrer sur le nœud 1. Ensuite, le thread T_0 écrit la valeur 7 dans la variable `x` du nœud 0. Enfin, les deux threads lisent et affichent la valeur locale de la variable `x` et le thread T_1 met fin à l'application par un appel à la primitive `pm2_halt`. Les deux threads se synchronisent par barrière (grâce à la primitive `pm2_thread_barrier`), de manière à s'assurer que les deux lectures ont bien lieu après l'écriture sur le nœud 0 et que la fin du programme est déclenchée après l'exécution des deux affichages.

L'utilisateur doit d'abord sélectionner les machines sur lesquelles le programme va s'exécuter, à l'aide de la commande `pm2_conf`. Ici, les machines `popc1` et `popc2` sont sélectionnées :

```
popc1% pm2conf popc1 popc2
The current PM2 configuration contains 2 host(s) :
0 : popc1
1 : popc2
```

Ensuite, le programme est chargé sur tous les nœuds sélectionnés à l'aide de la commande `pm2_load` :

```
popc1% pm2load exemple1
x = 7 sur le noeud 0
```

Afin de visualiser l'ensemble des sorties sur tous les nœuds et non seulement celles du nœud 0, on lance la commande `pm2_logs` :

```
popc1% pm2logs
*** Host popc1, process 0:
x = 7 sur le noeud 0

*** Host popc2, process 1:
x = 0 sur le noeud 1
```

Naturellement, les valeurs affichées sur les deux nœuds sont différentes, car chaque nœud possède sa propre copie de la variable et l'écriture sur la variable `x` située sur le nœud 0 n'affecte en rien la valeur de la variable `x` située sur le nœud 1.

3.5.2 Partage de données statiques

Le programme présenté sur la figure 3.9 illustre l'utilisation des données partagées statiques. Il est très similaire au programme précédent, mais ici la variable `x` est globalement partagée par les deux nœuds. Cette variable est maintenant déclarée à l'intérieur de la section du code délimitée par `BEGIN_DSM_DATA` et `END_DSM_DATA`. Afin d'assurer la cohérence des données partagées, un protocole de cohérence (ici, `li_hudak`) est sélectionné à l'aide de la primitive `dsm_set_default_protocol`.

Cette fois-ci, les résultats des deux lectures est le même :

```
popc1% pm2logs
*** Host popc1, process 0:
x = 7 sur le noeud 0

*** Host popc2, process 1:
x = 7 sur le noeud 1
```

L'exemple discuté plus haut illustre l'utilisation d'un protocole de cohérence séquentielle basé sur la réplication des pages en lecture (`li_hudak`). Le même programme (légèrement modifié) peut être exécuté en utilisant notre deuxième protocole de cohérence séquentielle basé sur la migration de threads : `migrate_thread`. C'est cette valeur qu'il faut fournir à la primitive `dsm_set_default_protocol`. Nous obtenons alors le code présenté sur la figure 3.10.

L'exécution de ce programme produira pourtant une sortie différente :

```
popc1% pm2logs
*** Host popc1, process 0:
x = 7 sur le noeud 0
x = 7 sur le noeud 1

*** Host popc2, process 1:
```

```
#include "pm2.h"

int x = 0;
pm2_thread_barrier_t b;


void f0 (void *arg)
{
  pm2_thread_barrier(&b);
  x = 7;
  pm2_thread_barrier(&b);
  tprintf("x = %d sur le noeud %d\n", x, pm2_self());
  pm2_thread_barrier(&b);
}

void f1 (void *arg)
{
  pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_thread_barrier(&b);
  tprintf("x = %d sur le noeud %d\n", x, pm2_self());
  pm2_thread_barrier(&b);
  pm2_halt();
}


int pm2_main (int argc, char **argv)
{
  pm2_thread_barrier_attr_t attr;

  /* Initialiser la barrière: 1 thread par noeud va y participer : */
  attr.local =  1;
  pm2_thread_barrier_init((&b, &attr);
  pm2_init (&argc, argv);
  if (pm2_self () == 0){
    /* Créer 2 threads qui vont exécuter f0 et f1 : */
    pm2_thread_create (f0, NULL);
    pm2_thread_create (f1, NULL);
  }
  pm2_exit ();
  return 0;
}
```

Figure 3.8 – Exemple de programme PM^2.

```
#include "pm2.h"

BEGIN_DSM_DATA
int x = 0;
END_DSM_DATA

pm2_thread_barrier_t b;


void f0 (void *arg)
{
  pm2_thread_barrier(&b);
  x = 7;
  pm2_thread_barrier(&b);
  tprintf("x = %d sur le noeud %d\n", x, pm2_self());
  pm2_thread_barrier(&b);
}

void f1 (void *arg)
{
  pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_thread_barrier(&b);
  tprintf("x = %d sur le noeud %d\n", x, pm2_self());
  pm2_thread_barrier(&b);
  pm2_halt();
}


int pm2_main (int argc, char **argv)
{
  pm2_thread_barrier_attr_t attr;

  dsm_set_default_protocol (li_hudak);
  /* Initialiser la barrière: 1 thread par noeud va y participer : */
  attr.local =  1;
  pm2_thread_barrier_init((&b, &attr);
  pm2_init (&argc, argv);
  if (pm2_self () == 0){
    /* Créer 2 threads qui vont exécuter f0 et f1 : */
    pm2_thread_create (f0, NULL);
    pm2_thread_create (f1, NULL);
  }
  pm2_exit ();
  return 0;
  }
```

Figure 3.9 – Déclaration de données partagées statiques et sélection d'un protocole de cohérence forte.

```
#include "pm2.h"

BEGIN_DSM_DATA
int x = 0;
END_DSM_DATA

pm2_thread_barrier_t b;


void f0 (void *arg)
{
  pm2_thread_barrier(&b);
  x = 7;
  pm2_thread_barrier(&b);
  tprintf("x = %d sur le noeud %d\n", x, pm2_self());
  pm2_thread_barrier(&b);
}

void f1 (void *arg)
{
  pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_thread_barrier(&b);
  tprintf("x = %d sur le noeud %d\n", x, pm2_self());
  pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_halt();
}

int pm2_main (int argc, char **argv)
{
  pm2_thread_barrier_attr_t attr;

  dsm_set_default_protocol (migrate_thread);
  /* Initialiser la barrière: 1 thread par noeud va y participer : */
  attr.local =  1;
  pm2_thread_barrier_init((&b, &attr);
  pm2_init (&argc, argv);
  if (pm2_self () == 0){
    /* Créer 2 threads qui vont exécuter f0 et f1 : */
    pm2_thread_create (f0, NULL);
    pm2_thread_create (f1, NULL);
  }
  pm2_exit ();
  return 0;
}
```

Figure 3.10 – Utilisation d'un protocole basé sur la migration de threads.

En effet, la variable `x` est stockée par défaut sur le nœud 0 et le protocole `migrate_thread` n'utilise ni réplication, ni migration de page. L'accès à la variable `x` déterminera la migration du thread T_1 du nœud 1 vers le nœud 0 et donc les deux affichages se feront sur ce même nœud.

Remarquons aussi qu'à la fin du programme le thread T_1 migre à nouveau sur le nœud 1 avant la dernière synchronisation. Ceci est exigé par l'implémentation actuelle de la barrière de synchronisation des threads, à laquelle doivent participer sur chaque nœud autant de threads qu'indiqué à l'initialisation de la barrière (ici, un thread par nœud). Une implémentation de la barrière qui n'impose pas cette restriction est en cours de développement.

3.5.3 Allocation dynamique de données partagées

Dans les exemples présentés plus haut, la variable partagée `x` est déclarée statiquement. DSM-PM^2 fournit également un mécanisme d'allocation dynamique de mémoire partagée, accessible par la primitive `pm2_malloc`. L'utilisation de cette primitive est illustrée dans le programme présenté sur la figure 3.11, obtenu en modifiant légèrement le programme 3.9. Cette primitive prend en entrée un *attribut d'allocation* qui permet d'indiquer le statut de la zone mémoire (dans ce cas, partagée : `ISO_SHARED`) et d'associer un protocole de cohérence à cette zone (dans cet exemple, `li_hudak`). L'utilisateur peut donc éventuellement associer des protocoles de cohérence différents à des données différentes lors de l'allocation des zones mémoires correspondantes. Le propriétaire initial de la variable sera le nœud qui effectue l'allocation (ici, le nœud 0).

Au lieu d'accéder directement à la variable partagée, les threads y accèdent ici à travers un pointeur. Remarquons qu'initialement seul le processus 0, qui effectue l'allocation, connaît l'adresse de l'entier partagé (`ptr`) et que cette adresse est copiée ensuite par chaque thread dans un pointeur local (`my_ptr`). Ainsi, en cas de migration, le thread connaîtra toujours l'adresse de la donnée partagée (ici, c'est le cas du thread qui exécute la fonction `f1`).

La sortie obtenue en exécutant ce programme est similaire à celle du programme de la figure 3.9 :

```
popc1% pm2logs
*** Host popc1, process 0:
valeur = 7 sur le noeud 0

*** Host popc2, process 1:
valeur = 7 sur le noeud 1
```

3.5.4 Sélection dynamique de protocole pour une application donnée

Une application DSM-PM^2 peut être exécutée en utilisant plusieurs protocoles de cohérence alternatifs qui implémentent le même modèle de cohérence. Le choix du protocole peut se faire à l'exécution, sans que cela nécessite la recompilation de l'application ! Ceci est illustré dans le programme présenté sur la figure 3.12. Ce programme obtenu par une légère modification du programme précédent, accepte un argument entier, qui détermine dynamiquement le protocole à utiliser : `li_hudak` si l'argument est nul, `migrate_thread` sinon.

Les sorties du programme dans les deux cas de figure sont données ci-dessous. Dans le cas du protocole `li_hudak`, l'affichage du thread T_1 se fait sur le nœud 1, alors que dans le cas du

```
#include "pm2.h"

pm2_thread_barrier_t b;
int *ptr;

void f0 (void *arg)
{
  int *my_ptr = ptr;

  pm2_thread_barrier(&b);
  *my_ptr = 7;
  pm2_thread_barrier(&b);
  tprintf("valeur = %d sur le noeud %d\n", *my_ptr, pm2_self());
  pm2_thread_barrier(&b);
}

void f1 (void *arg)
{
  int *my_ptr = ptr;

  pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_thread_barrier(&b);
  tprintf("valeur = %d sur le noeud %d\n", *my_ptr, pm2_self());
  pm2_thread_barrier(&b);
  pm2_halt();
}


int pm2_main (int argc, char **argv)
{
  pm2_thread_barrier_attr_t b_attr;

  /* Initialiser la barrière: 1 thread par noeud va y participer : */
  b_attr.local =  1;
  pm2_thread_barrier_init((&b, &b_attr);
  pm2_init (&argc, argv);
  if (pm2_self () == 0){
    /* Allocate shared variable: */
    isoaddr_attr_t alloc_attr;
    isoaddr_attr_set_status(&alloc_attr, ISO_SHARED);
    isoaddr_attr_set_protocol(&alloc_attr, li_hudak);
    ptr = (int *)pm2_malloc(sizeof(int), &alloc_attr);
    /* Créer 2 threads qui vont exécuter f0 et f1 : */
    pm2_thread_create (f0, NULL);
    pm2_thread_create (f1, NULL);
  }
  pm2_exit ();
  return 0;
}
```

Figure 3.11 – Allocation dynamique de données partagées.

```
#include "pm2.h"

pm2_thread_barrier_t b;
int *ptr;

void f0 (void *arg)
{
  int *my_ptr = ptr;

  pm2_thread_barrier(&b);
  *my_ptr = 7;
  pm2_thread_barrier(&b);
  tprintf("valeur = %d sur le noeud %d\n", *my_ptr, pm2_self());
  pm2_thread_barrier(&b);
}

void f1 (void *arg)
{
  int *my_ptr = ptr;

  pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_thread_barrier(&b);
  tprintf("valeur = %d sur le noeud %d\n", *my_ptr, pm2_self());
  if (pm2_self() !=1 )
     pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_halt();
}


int pm2_main (int argc, char **argv)
{
  pm2_thread_barrier_attr_t b_attr;
  int prot;

  /* Initialiser la barrière: 1 thread par noeud va y participer : */
  b_attr.local =  1;
  pm2_thread_barrier_init((&b, &b_attr);
  pm2_init (&argc, argv);
  if (pm2_self () == 0){
    isoaddr_attr_t alloc_attr;
    /* Sélectionner dynamiquement le protocole :*/
    if (argv[1] == 0) prot = li_hudak;
    else              prot = migrate_thread;

    isoaddr_attr_set_status(&alloc_attr, ISO_SHARED);
    isoaddr_attr_set_protocol(&alloc_attr, prot);
    /* Allouer la variable partagée : */
    ptr = (int *)pm2_malloc(sizeof(int), &alloc_attr);
    /* Créer 2 threads qui vont exécuter f0 et f1 : */
    pm2_thread_create (f0, NULL);
    pm2_thread_create (f1, NULL);
  }
  pm2_exit ();
  return 0;
}
```

Figure 3.12 – Sélection dynamique de protocoles.

protocole `migrate_thread` ce thread fera son affichage sur le nœud 0. En effet, le thread migrera sur le nœud 0 dès qu'il accédera la deuxième variable, dont le nœud 0 est propriétaire.

```
popc1% pm2load exemple5 0
valeur = 7 sur le noeud 0

popc1% pm2logs
*** Host popc1, process 0:
valeur = 7 sur le noeud 0

*** Host popc2, process 1:
valeur = 7 sur le noeud 1

popc1% pm2load exemple5 1
valeur = 7 sur le noeud 0
valeur = 7 sur le noeud 1

popc1% pm2logs
*** Host popc1, process 0:
valeur = 7 sur le noeud 0
valeur = 7 sur le noeud 1

*** Host popc2, process 1:
```

3.5.5 Applications multiprotocoles

Une application DSM-PM^2 peut manipuler des données partagées gérées par des protocoles de cohérence différents. Le programme de la figure 3.13 est similaire aux programmes précédents, mais il manipule *deux* variables allouées sur le nœud 0. Lors de l'allocation dynamique de chaque variable on spécifie le protocole de cohérence associé à travers l'attribut d'allocation : `li_hudak` pour la première, `migrate_thread` pour la deuxième. Le thread T_1 affichera d'abord la valeur de la première variable sur le nœud 1, car la lecture détermine la réplication de la page correspondante sur le nœud 1, conformément au protocole `li_hudak`. Dans un deuxième temps, le thread affichera la valeur de la deuxième variable sur le nœud 0, car la lecture de cette variable gérée par le protocole `migrate_thread` déclenchera la migration du thread sur ce nœud. Le résultat de l'exécution de ce programme est le suivant :

```
popc1% pm2load exemple6
valeur1 = 7 sur le noeud 0
valeur2 = 8 sur le noeud 0
valeur2 = 8 sur le noeud 0

popc1% pm2logs
*** Host popc1, process 0:
valeur1 = 7 sur le noeud 0
valeur2 = 8 sur le noeud 0
```

```
valeur2 = 8 sur le noeud 0

*** Host popc2, process 1:
valeur1 = 7 sur le noeud 1
```

3.5.6 Utilisation de protocoles de cohérence faible

Les exemples présentés jusqu'ici ont illustré des protocoles de cohérence séquentielle. L'utilisation de la cohérence à la libération est introduite par le programme présenté sur la figure 3.14.

Ce programme censé pouvoir s'exécuter sur un nombre arbitraire de nœuds utilise une variable partagée de type entier allouée par le nœud 0. Le processus 0, exécuté sur le nœud 0, crée `NB_NODES` $\times$ `NB_THREADS_PER_NODE` threads, qui sont ensuite distribués cycliquement sur les `NB_NODES` nœuds disponibles. Chaque thread incrémente `NB_ITERATIONS` fois la variable et ensuite affiche sa valeur avant et après cette incrémentation. Chaque thread protège ses accès à la variable partagée par le verrou `L`, qui est déclaré statiquement comme variable partagée et peut être accédé par tous les threads, sur tous les nœuds. Avant toute utilisation, le verrou est initialisé par la primitive `dsm_lock_init`. Le verrou est géré par un protocole de cohérence séquentielle (ici `li_hudak`).

Dans cet exemple, un protocole de cohérence à la libération (`hbrc_mw`) est associé à la variable partagée lors de son allocation. Ce protocole doit être explicitement associé également au verrou `L` lors de l'initialisation de ce dernier. Cela est réalisé à travers un attribut d'initialisation du verrou. Cette association est nécessaire pour permettre à DSM-PM^2 d'appeler implicitement les routines *acquire* et *release* du protocole `hbrc_mw` lors des opérations de blocage et de libération de verrou.

Un exemple d'exécution de cette application est donné ci-dessous.

```
popc1% pm2conf popc1 popc2 popc3
The current PM2 configuration contains 3 host(s) :
0 : popc1
1 : popc2
2 : popc3

popc1% pm2load exemple7
Thread 0 commence sur le noeud 0
Thread 0 du noeud 0 finit sur le noeud 0: de 0 a 10!
Thread 3 commence sur le noeud 0
Thread 3 du noeud 0 finit sur le noeud 0: de 30 a 40!

popc1% pm2logs
*** Host popc1, process 0:
Thread 0 commence sur le noeud 0
Thread 0 du noeud 0 finit sur le noeud 0: de 0 a 10!
Thread 3 commence sur le noeud 0
Thread 3 du noeud 0 finit sur le noeud 0: de 30 a 40!

*** Host popc2, process 1:
```

```
#include "pm2.h"

pm2_thread_barrier_t b;
int *ptr1, *ptr2;

void f0 (void *arg)
{
  int *my_ptr1 = ptr1, *my_ptr2 = ptr2;

  pm2_thread_barrier(&b);
  *my_ptr1 = 7;
  *my_ptr2 = 8;
  pm2_thread_barrier(&b);
  tprintf("valeur1 = %d sur le noeud %d\n", *my_ptr1, pm2_self());
  tprintf("valeur2 = %d sur le noeud %d\n", *my_ptr2, pm2_self());
  pm2_thread_barrier(&b);
}

void f1 (void *arg)
{
  int *my_ptr1 = ptr1, *my_ptr2 = ptr2;

  pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_thread_barrier(&b);
  tprintf("valeur1 = %d sur le noeud %d\n", *my_ptr1, pm2_self());
  tprintf("valeur2 = %d sur le noeud %d\n", *my_ptr2, pm2_self());
  if (pm2_self() !=1 )
     pm2_migrate_self(1);
  pm2_thread_barrier(&b);
  pm2_halt();
}


int pm2_main (int argc, char **argv)
{
  pm2_thread_barrier_attr_t b_attr;
  int prot;

  /* Initialiser la barrière: 1 thread par noeud va y participer: */
  b_attr.local =  1;
  pm2_thread_barrier_init((&b, &b_attr);
  pm2_init (&argc, argv);
  if (pm2_self () == 0){
    isoaddr_attr_t attr1, attr2;
    isoaddr_attr_set_status(&attr1, ISO_SHARED);
    isoaddr_attr_set_protocol(&attr1, li_hudak);
    ptr1 = (int *)pm2_malloc(sizeof(int), &attr1);
    isoaddr_attr_set_status(&attr2, ISO_SHARED);
    isoaddr_attr_set_protocol(&attr2, migrate_thread);
    ptr2 = (int *)pm2_malloc(sizeof(int), &attr2);
    /* Créer 2 threads qui vont exécuter f0 et f1 */
    pm2_thread_create (f0, NULL);
    pm2_thread_create (f1, NULL);
  }
  pm2_exit ();
  return 0;
}
```

Figure 3.13 – Utilisation de plusieurs protocoles au sein du même programme.

```
#include "pm2.h"

#define NB_NODES (pm2_config_size())
#define NB_THREADS_PER_NODE 2
#define NB_ITERATIONS 10

BEGIN_DSM_DATA;
dsm_lock_t L = NULL;
END_DSM_DATA;

int *ptr;

void f (void *arg)
{
  int i, initial, final;
  int *my_ptr = ptr;
  int thread_id = (int*)arg;
  int my_node = thread_id%NB_NODES;

  if (my_node != 0)
    pm2_migrate_self(my_node);
  tprintf ("Thread %d commence sur le noeud %d\n", thread_id, pm2_self());
  dsm_lock (L);
  initial = *my_ptr;
  for (i = 0; i < NB_ITERATIONS; i++)
    (*my_ptr)++;
  final = *my_ptr;
  dsm_unlock (L);
  tprintf ("Thread %d finit sur le noeud %d: "
   "de %d a %d!\n", thread_id, pm2_self (), initial, final);
}


int pm2_main (int argc, char **argv)
{
  dsm_lock_attr_t lock_attr;
  dsm_set_default_protocol (li_hudak);
  dsm_lock_attr_register_protocol(&lock_attr, hbrc_mw);
  dsm_lock_init(&L, &lock_attr);
  pm2_init (&argc, argv);
  if (pm2_self () == 0){
    isoaddr_attr_t alloc_attr;
    isoaddr_attr_set_status(&alloc_attr, ISO_SHARED);
    isoaddr_attr_set_protocol(&alloc_attr, hbrc_mw);
    ptr = (int *)pm2_malloc(sizeof(int), &alloc_attr);
    for (i = 0; i < NB_NODES *  NB_THREADS_PER_NODE; i++)
      pm2_thread_create (f, i);
  }
  pm2_exit ();
  return 0;
}
```

Figure 3.14 – Utilisation d'un protocole de cohérence faible.

```
int pm2_main (int argc, char **argv)
{
  ...
  dsm_lock_attr_t attr;

  prot = dsm_create_protocol(my_read_fault_handler ,
     my_write_fault_handler,
     my_read_server,
     my_write_server,
     my_invalidate_server,
     my_receive_page_server,
     my_acquire_func,
     my_release_func,
     my_prot_init_func
     );
  dsm_set_default_protocol (prot);
  pm2_init (&argc, argv);
  ...
}
```

Figure 3.15 – Définition d'un nouveau protocole.

```
Thread 1 commence sur le noeud 0
Thread 1 du noeud 0 finit sur le noeud 0: de 10 a 20!
Thread 4 commence sur le noeud 0
Thread 4 du noeud 0 finit sur le noeud 0: de 40 a 50!

*** Host popc3, process 2:
Thread 2 commence sur le noeud 2
Thread 2 du noeud 2 finit sur le noeud 2: de 20 a 30!
Thread 5 commence sur le noeud 0
Thread 5 du noeud 0 finit sur le noeud 0: de 50 a 60!
```

3.5.7 Définir de nouveaux protocoles

Les exemples précédents ont montré l'utilisation des protocoles intégrés dans la plate-forme. De nouveaux protocoles peuvent être proposés par l'utilisateur et employés de la même manière. Dans le programme illustré sur la figure 3.15, un nouveau protocole est défini en fournissant 9 routines à la primitive `dsm_create_protocol`. L'identifiant obtenu (`prot`) est ensuite utilisé au même titre que les identifiants de protocoles intégrés (`li_hudak`, `hbrc_mw`, etc) dans les exemples précédents.

3.6 Conclusion

Chaque effort de recherche dans le domaine des MVP a été traditionnellement validé par la construction d'un nouveau système, afin d'illustrer un nouveau concept, modèle ou protocole de cohérence. Même si dans ce nouveau système seulement peu de mécanismes sont effectivement dédiés au concept illustré, et même si la plupart des composants du nouveau système repose

souvent sur des mécanismes classiques de base, un effort important a dû être investi à chaque fois pour construire le système complet.

La proposition développée dans ce chapitre fournit un cadre unifié permettant l'implémentation et l'expérimentation facile de protocoles de cohérence pour la mémoire virtuellement partagée. Nous avons identifié un ensemble de mécanismes de base constituant ce que nous appelons une *MVP générique* et que le concepteur de protocole peut utiliser pour rapidement expérimenter des protocoles de cohérence. Cette notion a été dégagée suite à une étude restreinte à quelques MVP existantes, utilisant une cohérence à base de pages et implémentées entièrement au niveau logiciel, en espace utilisateur (donc sans aucune modification du système d'exploitation sous-jacent). Ceci renforce la *portabilité* de la MVP proposée, propriété nécessaire dans un contexte expérimental tel que le nôtre.

La plate-forme que nous proposons fournit des mécanismes permettant l'implémentation de protocoles pour plusieurs modèles de cohérence. À titre d'exemple, nous avons développé 2 protocoles pour la cohérence séquentielle, 2 pour la cohérence à la libération et 2 pour la cohérence Java. À travers ces 2 derniers, nous avons illustré l'utilisation de la plate-forme DSM-PM^2 en tant que cible d'un système de compilation Java pour grappes. Afin d'élargir la généricité de DSM-PM^2, nous nous sommes proposé de réaliser un travail similaire pour le langage OpenMP.

Une caractéristique importante du support générique de la plate-forme, ainsi que des protocoles de cohérence intégrés dans sa bibliothèque, est le fait qu'ils sont fonctionnels dans un contexte *multithread*. Plusieurs threads peuvent exécuter des actions de cohérence de manière concurrente et cette concurrence rend la conception des protocoles plus complexe. Ces aspects ont été brièvement illustrés pour plusieurs protocoles de la bibliothèque. Une présentation plus technique de certains points est faite dans le chapitre suivant.

Chapitre 4

Implémentation et performance

Ce chapitre présente quelques aspects liés à l'implémentation de DSM-PM^2 et fournit une évaluation préliminaire des performances de la plate-forme. Dans un premier temps, nous abordons le problème de la gestion de la concurrence des actions de cohérence (section 4.1). La prise en compte de cette *concurrence* dans les protocoles est nécessaire afin d'exploiter efficacement le *parallélisme* potentiel dans un contexte multithread et rend ces protocoles plus complexes que les protocoles classiques. Une évaluation expérimentale préliminaire du coût des opérations de base utilisées par ces protocoles est donnée dans la section 4.2.

Un autre aspect important de la plate-forme concerne l'allocation dynamique des données partagées. Cette allocation se fait suivant une approche *iso-adresse*, permettant aux pages d'être toujours stockées à la même adresse virtuelle lors des migrations ou des réplications. Cette approche est nécessaire afin d'assurer la validité globale des pointeurs vers ces données et permet également de mettre en œuvre des protocoles basés sur la migration des threads, comme nous l'avons montré dans le chapitre précédent. Dans la section 4.3 nous présentons la version étendue de l'allocateur iso-adresse de PM^2 introduit dans la section 2.2.3. Cet allocateur permet désormais l'allocation de données partagées.

Les deux sections suivantes sont consacrées à la comparaison des performances globales obtenues au niveau applicatif lorsqu'on utilise différents protocoles de cohérence de la plate-forme. L'utilisation de stratégies différentes (migration de threads *vs* transfert de pages) est discutée sur un cas simple d'application qui résout le problème du voyageur de commerce (section 4.4). La section 4.5 présente une autre série d'expériences effectuées sur une application typique pour les systèmes à MVP : un calcul de transformée de Fourier 1D. Cette application a été extraite de l'ensemble d'applications SPLASH-2 [102], communément utilisé pour l'évaluation des MVP, et a été adaptée pour une exécution multithread au-dessus DSM-PM^2. Les expériences effectuées ont mis en évidence les performances relatives de quelques protocoles intégrés dans la bibliothèque de DSM-PM^2 et d'étudier l'influence du multithreading sur ces performances sur des architectures à caractéristiques différentes.

Afin d'évaluer l'utilisation de DSM-PM^2 comme cible des systèmes de compilations de langages basés sur la mémoire partagée, nous avons également effectué des expériences sur plusieurs applications multithreads Java, compilées avec Hyperion. Ces expériences, qui ont eu pour objectif de comparer les performances de nos deux protocoles qui implémentent la cohérence Java, font l'objet d'une description détaillée dans le chapitre 5, dédié à l'implémentation d'Hyperion au-dessus de DSM-PM^2.

```
void read_fault_handler(unsigned int index)
{
       int reply_node;

       dsm_lock_page(index);

       dsm_send_page_req_and_wait(dsm_get_owner(index), index, dsm_self(),
           READ_ACCESS, &reply_node);
       dsm_set_owner(index,reply_node);
       dsm_set_access(index, READ_ACCESS);

       dsm_unlock_page(index);
}
```

Figure 4.1 – Routine de gestion des défauts de page en lecture dans le protocole original de Li et Hudak.

4.1 Gestion de la concurrence dans les protocoles de cohérence multithread

Dans un système multithread à mémoire virtuellement partagée, plusieurs threads peuvent accéder les *mêmes* données, sur le *même* nœud, quasiment en *même* temps. Ces accès peuvent déclencher des défauts de page concurrents, qui nécessitent l'exécution d'actions de cohérence. Une approche possible consiste à *sérialiser* sur chaque nœud les actions de cohérence concernant la même page. Cette approche appliquée traditionnellement dans les MVP monothreads a l'avantage de la simplicité, mais a aussi l'inconvénient de limiter le parallélisme potentiel des actions de cohérence et donc de réduire implicitement les recouvrements potentiels générateurs d'efficacité : pendant qu'un thread attend une page, un autre thread pourrait s'exécuter. Afin de tirer profit des avantages du multithreading, un degré de concurrence maximal de ces traitements est souhaitable. Nous avons évoqué dans la section 3.4 plusieurs situations de concurrence qui doivent être gérées par une MVP multithread. Dans cette section nous présenterons de manière plus détaillée quelques situations de ce type.

4.1.1 Défauts de page et concurrence

Traditionnellement, dans les MVP monothreads, les défauts de page sont traités de manière *séquentielle* sur chaque nœud. En effet, il ne peut y avoir de défauts de page concurrents sur un même nœud, puisqu'il n'y a qu'un seul flot d'exécution local. De plus toute requête de page qui arrive sur un nœud en train de traiter un défaut de page sera mise en attente jusqu'à ce que ce traitement soit terminé. En bref, les routines qui composent le protocole de cohérence sont exécutées de manière *atomique*. L'implémentation de cette atomicité repose le plus souvent sur des sections critiques.

Sur la figure 4.1, nous présentons un code C qui correspondrait à la routine de gestion d'un défaut de page en lecture, suivant l'algorithme du gestionnaire dynamique distribué proposé par Li et Hudak [64]. Le traitement est simple : le processus qui a provoqué le défaut de page envoie une requête au propriétaire probable de la page et ensuite attend la page. Une fois la page arrivée, le processus met à jour deux informations dans la table locale des pages : le nœud propriétaire

```
void ask_for_read_copy(unsigned int index)
{
  dsm_lock_page(index);

  if (dsm_get_access(index) != NO_ACCESS)
    {
      /* Un autre thread local a demandé la page et celle-ci est déjà là. */
      dsm_unlock_page(index);
      return;
    }

  if (dsm_get_pending_access(index) == NO_ACCESS)
    {
      /* Aucune requête ne vient d'être envoyée pour cette page */
      dsm_set_pending_access(index, READ_ACCESS);
      dsm_send_page_req(dsm_get_owner(index), index, dsm_self(), READ_ACCESS);
    }
  /* sinon, une requête a déjà été envoyée et il suffit d'attendre la page */
  dsm_wait_for_page(index);

  dsm_unlock_page(index);
}
```

Figure 4.2 – Routine de gestion des défauts de page en lecture pour le protocole `li_hudak`.

(`reply_node`) et le type d'accès (`READ_ACCESS`). Nous rappelons que, dans le cas du gestionnaire dynamique, le propriétaire de la page peut être différent du nœud auquel la requête a été envoyée et que donc l'identité du propriétaire courant est transmise en même temps que la page, afin de permettre l'actualisation de la table des pages sur le nœud originaire de la requête.

Nous remarquerons tout d'abord que le code est délimité par une section critique, interdisant tout autre manipulation de l'entrée correspondante de la table des pages pendant le traitement. Même si dans ce contexte un seul thread de niveau applicatif s'exécute sur chaque nœud, nous pouvons supposer qu'un processus démon puisse tourner en parallèle pour gérer les communications du nœud. Ce démon peut servir les requêtes de pages et doit, à cet effet, accéder/modifier la table locale des pages. Le rôle de la section critique est d'empêcher une éventuelle concurrence entre les accès de ce processus et ceux du thread de niveau applicatif.

Remarquons également que la requête de page est synchrone : le processus est bloqué jusqu'à la réception de la page. La suite du code représente le traitement après réception. Ainsi, le code correspond en fait à *deux* événements et regroupe les actions correspondant au *défaut de page en lecture*, d'une part, et à la *réception de la page*, d'autre part.

Dans notre définition de protocole de cohérence multithread, ces deux événements sont traités par des routines différentes : la *routine de gestion du défaut de page en lecture* et la *serveur de réception des pages*. Le code de ces routines, conçu pour un contexte multithread, est présenté sur les figures 4.2 et respectivement 4.3. Dans le souci d'augmenter le degré de concurrence permis dans un contexte multithread, les deux traitements sont séparés et ne sont pas exécutés à l'intérieur de la même section critique, mais font l'objet de deux sections critiques différentes. En effet, une section critique unique limiterait sévèrement le parallélisme potentiel. Il n'y a aucune raison pour que d'autres threads ayant besoin d'exécuter des actions de cohérence sur la même page restent bloqués entre le moment où la requête de page est envoyée et la réception de la page. La prise

en compte de ces possibles situations de concurrence rend nos routines multithreads bien plus complexes que la routine initiale. Voici, à titre d'exemple, le cas le plus complexe de concurrence qui peut se produire lors de l'exécution concurrente de la routine de gestion des défauts de page en lecture `ask_for_read_copy` du protocole `li_hudak`.

Supposons qu'un thread T_1 accède en lecture une page P, alors que la page n'est accessible ni en lecture, ni en écriture sur le nœud local. Le thread entrera dans la routine de gestion du défaut de page en lecture et enverra une requête au propriétaire probable de la page P et ensuite se bloquera en attente de cette page. Ce blocage libère implicitement le verrou associé à la page. Supposons maintenant qu'un autre thread T_2 essaye de lire la page : il déclenchera un défaut de page et entrera à son tour dans la routine de gestion du défaut de page. Mais ce thread n'a pas besoin d'envoyer une deuxième requête, il suffira qu'il attende la réception de la page P, en réponse à la requête du thread T_1. Le rôle du test

```
if (dsm_get_pending_access(index)) == NO_ACCESS)
```

dans la routine `ask_for_read_copy` (figure 4.2) est justement de détecter si une requête a déjà été envoyée et de permettre de factoriser la requête précédente. La primitive `dsm_get_pending_access(index)` retourne `NO_ACCESS` si aucun thread ayant précédemment envoyé une requête n'est en train d'attendre sa réponse. Dans le cas contraire, elle retourne `READ_ACCESS` ou `WRITE_ACCESS`, suivant le type de requête. Le droit d'accès en écriture implique également le droit d'accès en lecture. Remarquons que la condition n'est pas :

```
if (dsm_get_pending_access(index)) != READ_ACCESS)
```

L'explication est simple : dans le cas où une requête *en écriture* vient d'être envoyée, la page qui sera reçue pourra également satisfaire la requête en lecture qui a déclenché le défaut de page en cours de traitement.

Enfin, supposons qu'un autre thread T_3 produise un défaut de page en lecture sur la même page et qu'au moment où il rentre dans la routine de gestion du défaut de page, la page arrive. Supposons que le thread qui exécute le serveur de réception des pages obtienne le verrou le premier, qu'il valide la page (`dsm_set_access(index,READ_ACCESS)`) et qu'il réveille les threads T_1, et T_2 bloqués. Ces threads pourront sortir de la routine de gestion du défaut de page en lecture, alors que le thread T_3 y rentrera à peine. Mais le nœud a déjà obtenu le droit d'accès en lecture et donc il n'y a plus rien à faire, sinon quitter la routine. C'est le rôle du test

```
if (dsm_get_access(index)) != NO_ACCESS)
```

au début de la routine `ask_for_read_copy`, la présence duquel peut étonner au premier abord.

4.1.2 Cohérence à la libération et concurrence

D'autres cas de concurrence interviennent lors des actions de cohérence déclenchées par les opérations de synchronisation (*acquire* et *release*) dans les protocoles de cohérence faible. Nous présentons dans la figure 4.4 le code de la routine *release* du protocole `hbrc_mw` (légèrement simplifié, pour des raisons de lisibilité). Ce code est exécuté automatiquement par DSM-PM^2 lors de la libération d'un verrou auquel le protocole `hbrc_mw` a été associé.

```
void li_hudak_validate_page(void *addr, dsm_access_t access, int reply_node)
{
     unsigned long index = dsm_page_index(addr);
     int node;

     dsm_lock_page(index);

     if (access == READ_ACCESS)
       {
         /* Valider l'accès à la page */
         dsm_set_access(index, READ_ACCESS);
         if (dsm_get_pending_access(index) == READ_ACCESS)
          {
            dsm_set_owner(index, reply_node);
            dsm_set_pending_access(index, NO_ACCESS);
          }
       }
     else /* access = WRITE_ACCESS */
       {
        ...
       }

     /* Réveiller les threads en attente de la page */
     dsm_signal_page_ready(index);
     ...
     dsm_unlock_page(index);
}
```

Figure 4.3 – Routine de réception des pages pour le protocole `li_hudak` (extrait).

```
void dsmlib_hbrc_release ()
{
  if(!dsm_test_and_set_pending_release_or_wait())
    {
      unsigned int index;

      index = remove_first_from_list (&page_list);
      while (index != NULL) {
        dsm_lock_page(index);
        if (dsm_get_owner (index) == dsm_self ())
          dsm_invalidate_copyset (index);
        else
          {
            if (diffs_to_send(index))
              {
                dsm_send_diffs (index, dsm_get_owner (index));
                dsm_set_access (index, NO_ACCESS);
                dsm_free_twin (index);
                dsm_send_invalidate_req (dsm_get_owner (index), index);
                dsm_wait_ack (index);
              }
  }
        dsm_unlock_page(index);
        index = remove_first_from_list (&page_list);
      }
      dsm_clear_pending_release_and_broadcast_done();
    }
}
```

Figure 4.4 – Gestion de la concurrence dans la routine *release* du protocole `hbrc_mw`.

La boucle de cette fonction parcourt la liste des pages modifiées depuis la dernière entrée en section critique. Pour chaque page, si le nœud local est le nœud de référence de la page, des invalidations sont envoyées à tous les nœuds qui possèdent une copie de la page. Sinon, la fonction calcule les *différences* correspondant aux modifications de la page en la comparant à sa copie jumelle et envoie ces différences au nœud de référence de la page. Dès que ce nœud confirme la réception des différences, le même traitement commence pour la page suivante.

L'implémentation multithread du protocole `hbrc_mw` doit garantir une exécution correcte des appels concurrents de cette routine par des threads différents. Une telle situation de concurrence est détectée par la fonction `dsm_test_and_set_pending_release_or_wait`. Si un thread appelle la routine *release* alors qu'un autre est en train de l'exécuter et a déjà envoyé les différences, le deuxième thread n'a plus besoin de les envoyer ; par contre, il devra attendre que la réception des différences envoyées par le premier thread soit confirmée avant de continuer son traitement.

Un autre cas de concurrence se produit si la routine *release* est appelée quasi-simultanément par deux threads sur deux nœuds différents, N_1 et N_2. Si les threads ont modifié une même page, ils vont devoir envoyer leurs différences vers le nœud de référence de la page (N_3). En recevant les premières différences (par exemple du nœud N_1), le nœud de référence enverra une requête d'invalidation au nœud N_2. Ce nœud doit alors répondre en envoyant ses différences au nœud N_3. Indépendamment de ceci, l'exécution de la routine *release* sur le nœud N_2 détermine aussi l'envoi des mêmes différences vers N_3. Notre implémentation garantit que les différences sont envoyées une seule fois, tout en permettant aux deux opérations (*release* et invalidation) de s'exécuter de manière concurrente. C'est le rôle du test `if (diffs_to_send(index))`.

4.1.3 Transfert des pages et concurrence

Un autre problème posé par la concurrence se produit lors de la réception d'une page sur un nœud. Considérons la situation suivante.

Supposons qu'un thread T_1 accède (en lecture ou en écriture) une page P et que cet accès déclenche un défaut de page. Le thread T_1 envoie une requête au nœud propriétaire de la page et ensuite se bloque. Lors de la réception de la page, un thread est créé pour exécuter le serveur de réception des pages. Ce thread doit écrire le contenu à jour de la page en mémoire et a donc besoin, à cet effet, de rendre temporairement modifiable la zone mémoire correspondante. Or, ceci peut mener à une violation des contraintes de cohérence : en effet, un autre thread T_2 de l'application, s'exécutant sur le même nœud, peut aussi accéder la page *pendant sa mise à jour* et lire des valeurs incohérentes, car la page n'est plus protégée. En effet, c'est seulement après l'exécution du serveur de réception des pages que la cohérence des données contenues par la page est garantie. Il est donc nécessaire d'empêcher tout thread de l'application d'accéder à la page avant que *toute* la zone mémoire correspondante soit à jour. Cette mise à jour doit apparaître aux threads comme *atomique*.

Pour résoudre ce problème, nous avons créé un mécanisme permettant au thread chargé de mettre à jour la page d'effectuer cette opération sans pour autant rendre la page accessible aux autres threads. Nous avons, pour cela, utilisé un système de *double projection en mémoire* d'un fichier associé à la page (figure 4.5). Lors de la réception de la page, la zone mémoire à écrire est associée à un fichier temporaire à l'aide de la primitive Unix `mmap`. Cette association permet aussi de protéger la page de tout accès en lecture ou en écriture. Ensuite, le même fichier est projeté dans une zone mémoire spéciale, à une adresse virtuelle réservée, cette fois-ci en autorisant l'accès en

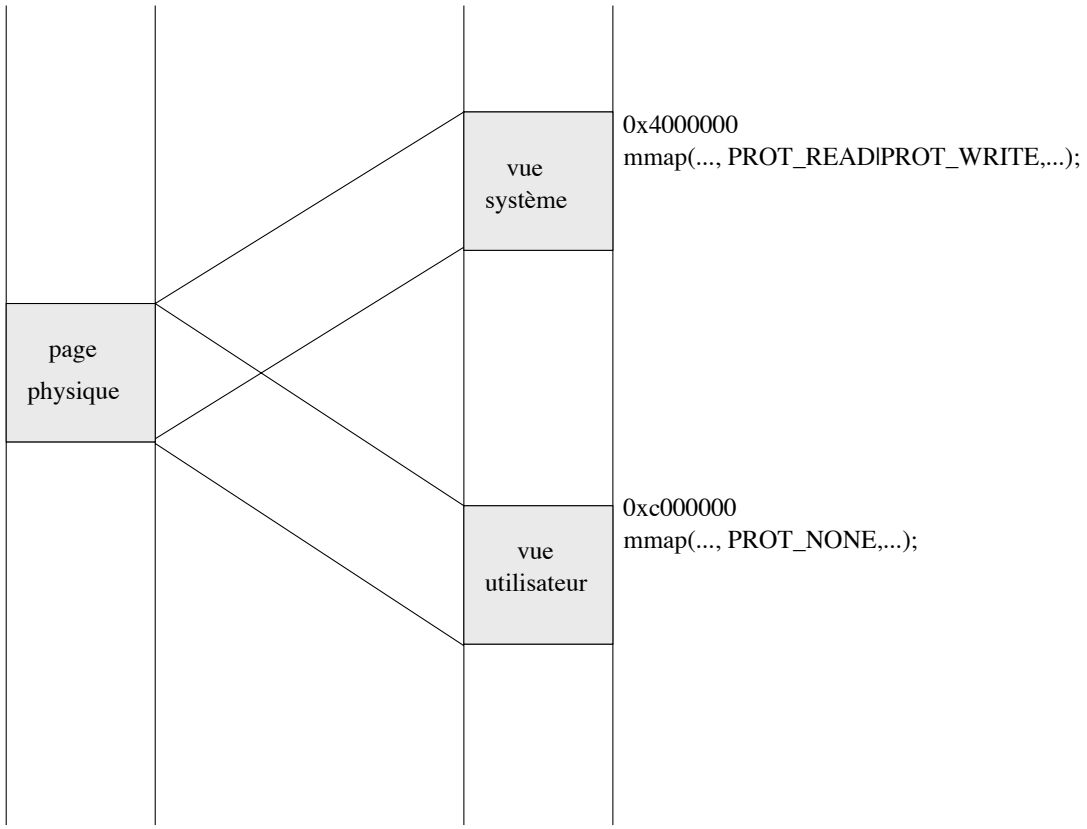


FIG. 4.5 – Lors de la réception d'une page sur un nœud, une double projection de la même page physique à deux adresses virtuelles est effectuée, avec des droits d'accès différents. Le thread interne de DSM-PM^2 chargé d'installer la page pourra ainsi la copier en mémoire en utilisant la "vue privée", accessible en écriture, alors que les threads de l'application qui utilisent la "vue publique" n'ont aucun droit d'accès à la page avant que la copie soit terminée.

écriture. La page est ensuite écrite en mémoire en utilisant cette deuxième adresse. À la fin de l'opération, le contenu de la page sera à jour, alors que l'intervalle d'adresses virtuelles utilisé par les threads de l'application est resté protégé. C'est seulement au moment où le serveur de réception des pages validera la page que les threads de l'application pourront y accéder. La cohérence des données accédées est ainsi garantie.

Nous avons mesuré le coût des opérations nécessaires pour réaliser la double projection sur nos grappes de PC à nœuds monoprocesseurs PII à 450 MHz. Ce coût est relativement élevé : 185 μs. Bien que coûteuse, cette solution permet de garantir l'atomicité de la réception des pages. Une meilleure efficacité serait sans doute obtenue si l'interface de programmation du système permettait d'associer explicitement des pages virtuelles avec des droits d'accès différents à la même page physique. Le mécanisme serait similaire à celui que nous avons implémenté, mais le double mapping se ferait à l'aide d'une clé logique qui identifie la page physique au lieu d'utiliser un fichier. Une telle interface n'est pas actuellement disponible sous Linux, mais ce serait une contribution utile à réaliser dans l'avenir.

Grappe	1	2	3	4
Processeur	PII 450 MHz	PII 450 MHz	PII 450 MHz	PII 450 MHz
Système d'exploitation	Linux 2.2.13	Linux 2.2.13	Linux 2.2.13	Linux 2.2.13
Protocole/Interface de comm.	BIP	TCP	TCP	SISCI
Réseau	Myrinet	Myrinet	Fast Ethernet	SCI

TAB. 4.1 – Description des grappes utilisées pour l'évaluation de la plate-forme.

4.2 Évaluation des mécanismes de base

Afin de donner une évaluation préliminaire des mécanismes de base de la plate-forme DSM-PM^2, nous avons mesuré les coûts des opérations de base les plus communes, sur différentes grappes dont les caractéristiques sont données dans le tableau 4.1.

4.2.1 Coût des accès générateurs de défauts de page

Dans la table 4.2 nous présentons la décomposition du coût d'un accès en lecture à une page absente de la mémoire locale, en supposant que le protocole correspondant soit basé sur le transfert des pages en réponse au défaut de page déclenché par l'accès. C'est le cas des protocoles `li_hudak`, `erc_sw`, `hbrc_mw`, qui utilisent la même routine de gestion des défauts de page en lecture (`ask_for_read_copy`, présentée dans la section 4.1.1)). Cette routine inspecte la table locale des pages partagées pour déterminer le nœud propriétaire de la page, ensuite envoie une *requête de page* à ce nœud. En réponse à la requête, le propriétaire envoie une copie de la page au nœud demandeur (*transfert de page*). Le temps de transfert présenté correspond à une page de 4 ko. Nous remarquerons que l'efficacité des communications influence de manière très significative le coût total de l'accès. Les mesures illustrées ici mettent en évidence la supériorité des protocoles BIP et SISCI par rapport à TCP en latence et en débit. Enfin, le *surcoût du protocole* comprend le temps de traitement de la requête sur le nœud propriétaire et le temps d'installation de la page sur le nœud demandeur. Ce surcoût correspond essentiellement à des modifications de la table des pages partagées et à des modifications des droits d'accès à la page, à l'aide de la primitive Unix `mprotect`. Le *surcoût de la double projection* indiqué ici montre l'influence des opérations effectuées pour garantir l'atomicité de la réception de la page (décrites dans la section 4.1.3) sur le coût global de l'accès. Nous avons pu remarquer que le coût de ces opérations, mesuré à 185 μs sur toutes les grappes utilisées, a une influence moins importante sur le coût global de l'accès : 50 à 175 μs. Cette atténuation pourrait être expliquée par des effets de cache. Une analyse plus fine serait pourtant nécessaire pour expliquer ce comportement.

Dans la table 4.3 nous présentons le coût (en μs) pour le traitement d'un défaut de page par un protocole basé sur la migration des threads, tel que `migrate_thread`. Le *surcoût du protocole* est ici très peu important (inférieur à 1 μs), car il est réduit à un appel à la primitive de migration de threads de PM^2, qui déplace le thread sur le nœud propriétaire. Cette opération comprend le transfert de la pile et du descripteur du thread et leur installation en mémoire sur le nœud destinataire. Le temps indiqué a été mesuré à partir du moment le thread est interrompu sur le nœud d'origine jusqu'au moment où le thread reprend son exécution sur le nœud destinataire. Il a été calculé en moyennant sur 5000 aller-retours. Dans ce test, la taille des données à transférer est de l'ordre de 1 kilooctet.

Opération (μs)	BIP/Myrinet	TCP/Myrinet	TCP/Fast Ethernet	SISCI/SCI
Défaut de page	11	11	11	11
Transmission requête	23	220	220	38
Transfert de page	138	343	736	119
Surcoût du protocole	26	26	26	26
Surcoût de la double projection	non mesurable	175	50	150
Coût total de l'accès (μs)	**>198**	**775**	**1043**	**344**

Nota : Pour des raisons liées à l'implémentation de BIP (et en particulier à son interaction avec le système de protection des pages mémoire), la technique du double mapping n'est pas utilisable avec la version courante de ce protocole. Ce problème sera résolu dans les prochaines versions de BIP. Ceci dit, un fonctionnement correct est garanti même si le double mapping est désactivé, dans le cas où les applications exécutent un seul thread applicatif par nœud.

TAB. 4.2 – Décomposition du coût du traitement d'un défaut de page en lecture dans un protocole basé sur le transfert de page.

Opération (μs)	BIP/Myrinet	TCP/Myrinet	TCP/Fast Ethernet	SISCI/SCI
Défaut de page	11	11	11	11
Migration du thread	75	280	373	62
Surcoût du protocole	1	1	1	1
Coût total de l'accès (μs)	**87**	**292**	**385**	**74**

TAB. 4.3 – Décomposition du coût de traitement d'un défaut de page en lecture dans un protocole basé sur la migration de thread.

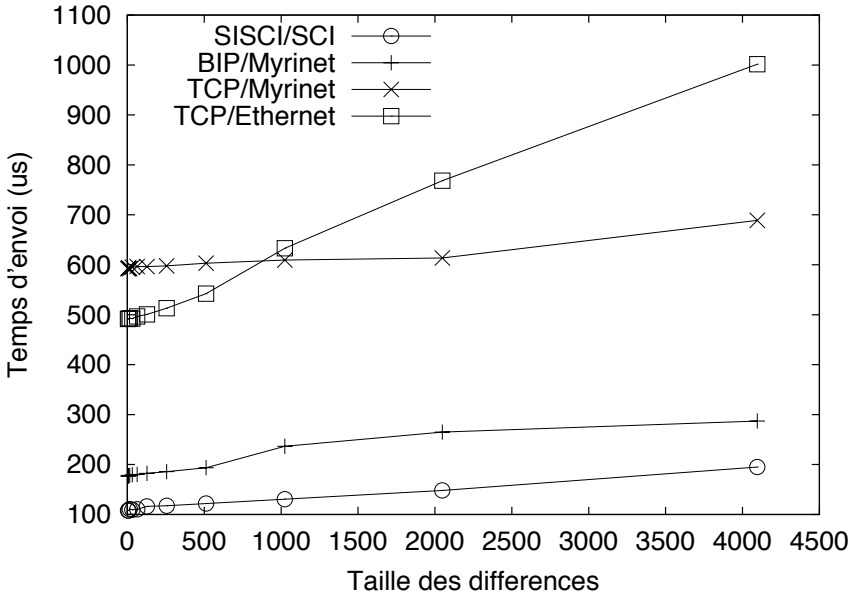


FIG. 4.6 – Coût de transmission des différences sur plusieurs types de grappes reliés par des réseaux différents. Les nœuds sont des monoprocesseurs PII 450 MHz.

Nous remarquerons que la stratégie basée sur la migration de thread est plus performante. L'explication en est multiple : d'abord, la migration comporte l'envoi d'un seul message, alors que le transfert de la page en comporte deux (requête + page) ; ensuite, la quantité de données à transférer est souvent inférieure dans les applications que nous avons rencontrées ; enfin, le surcoût rajouté par le protocole de cohérence au coût des communications est très faible. Néanmoins, il faut noter que le coût de la migration est très dépendant de la taille de la pile du thread. Dans notre programme de test, cette taille était réduite à 1 kilooctet, ce qui n'est pas forcément le cas dans toutes les applications. De plus, l'efficacité *globale* de ce type de protocole dépend également d'autres critères, tels que la distribution des données partagées sur les différents nœuds et les schémas d'accès des threads à ces données. En effet, étant donné que les threads migrent vers les données pour les accéder, ces deux facteurs ont une influence importante sur la distribution de la charge et donc sur l'efficacité. Ceci est illustré par nos expérimentations présentées dans la section 4.4. Les critères de choix entre un protocole basé sur le transfert des pages et un protocole basé sur la migration des threads doivent donc faire l'objet d'une analyse assez complexe. Nous n'avons pas abordé ce sujet dans le cadre de notre recherche, mais la plate-forme que nous proposons peut servir de support pour des expérimentations dans cette direction.

4.2.2 Coût de l'envoi des différences

Dans le cas des protocoles de cohérence faible à écrivains multiples, tels que `hbrc_mw`, l'opération *release* effectuée sur un nœud déclenche l'envoi des modifications (*différences*) effectuées par ce nœud sur une page vers le nœud de référence de la page.

Nous avons mesuré le coût de l'envoi des différences sur les 4 grappes à nœuds monoprocesseurs PII à 450 MHz, construites avec différentes interfaces de communication et différents

réseaux : SISCI/SCI, BIP/Myrinet, TCP/Myrinet, TCP/Ethernet. La figure 4.6 présente ce coût pour différentes tailles de ces différences, sur toutes ces plates-formes. Pour ces mesures, nous avons considéré le cas simple dans lequel les modifications effectuées sur la page sont regroupées sur une portion contiguë de la page. L'envoi comporte donc toujours un seul message. Le coût mesuré comprend l'envoi des différences, la mise à jour du nœud de référence, ainsi que la transmission du message d'acquittement en fin d'opération. Les coûts mesurés dépendent directement des performances des réseaux sous-jacents en termes de latence et débit. Nous remarquerons que dans tous les cas ils sont de l'ordre d'une centaine à plusieurs centaines de microsecondes.

4.3 Allocation iso-addresse pour mémoire virtuellement partagée

Un autre problème posé lors de la conception de DSM-PM^2 concerne l'allocation *dynamique* des données partagées. Tout comme les systèmes traditionnels à mémoire virtuellement partagée, DSM-PM^2 comprend un mécanisme d'allocation dynamique. Dans cette section, nous présentons les principes qui ont guidé la conception de cet allocateur que nous avons intégré dans le support générique de la plate-forme.

4.3.1 Iso-addresse et MVP

L'environnement PM^2 intégrait un allocateur dynamique de mémoire (`isomalloc`), que nous avons introduit dans la section 2.2.3. Cette allocateur, conçu dans la première partie de cette thèse, a eu pour objectif de permettre la migration des données tout en préservant la validité des pointeurs vers ces données, grâce à une *approche iso-adresse* : tout appel à la primitive d'allocation retourne un intervalle d'adresses virtuelles disponibles sur tous les nœuds. Par conséquent, en cas de migration, les données stockées dans la zone de mémoire correspondante peuvent être logées à la même adresse virtuelle sur le nœud destinataire que sur le nœud de départ. Il s'agit des données locales, stockées dans la pile du thread, mais aussi de données privées du thread allouées dynamiquement, censées le suivre en cas de migration. Tout pointeur utilisé par le thread migré pour accéder ces données reste donc valide, sans aucun traitement post-migration, quelle que soit la localisation courante du thread.

Une manière simple, mais coûteuse, d'assurer cette cohérence *globale* serait de réaliser une synchronisation de l'ensemble des nœuds lors de chaque allocation, pour "négocier" l'allocation d'un intervalle d'adresses virtuelles disponible sur tous les nœuds. L'allocateur iso-adresse de PM^2 propose une stratégie plus efficace qui permet à tout nœud d'allouer localement des zones mémoires qui lui sont réservées, sans engendrer des communications avec les autres nœuds au moment de l'allocation dans la plupart des cas. Une zone spéciale de l'espace virtuel d'adressage est réservée sur tous les nœuds pour les allocations iso-adresse (figure 4.7). Cette zone est divisée en intervalles disjoints de taille fixe (appelés *slots*) et ces intervalles sont préaffectés aux différents nœuds. Chaque nœud peut *allouer localement* de la mémoire dans les slots qui lui sont *globalement réservés*. Une négociation globale n'est nécessaire qu'en cas de pénurie de slots sur le nœud.

La principale application immédiate d'`isomalloc` a été son utilisation dans des régulateurs dynamiques de charge basés sur la migration *préemptive* et *transparente* des threads. Nous avons validé cette approche au sein de deux compilateurs de langages à parallélisme de données, C* et HPF (travail en collaboration avec Christian Pérez). Dans ce contexte, chaque processeur abstrait

du langage est projeté sur un thread PM^2 et traite un sous-ensemble des données de l'application. Les données sont donc associées aux threads et les suivent en cas de migration.

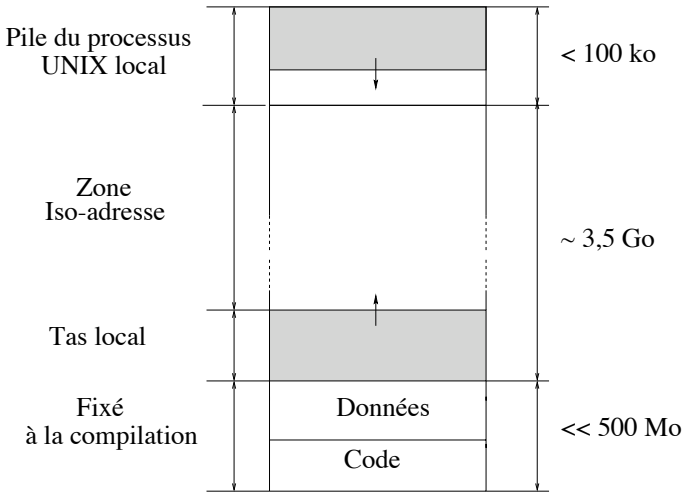


FIG. 4.7 – Sur chaque nœud, PM^2 lance un seul processus Unix exécutant le même code binaire. La cartographie mémoire des nœuds est identique ; en particulier, la zone iso-adresse (située entre la pile et le tas du processus) correspond au même intervalle d'adresses virtuelles sur tous les nœuds.

Dans un environnement à mémoire virtuellement partagée, le schéma d'accès des threads aux données est beaucoup plus complexe. Les données ne sont plus attachées à des threads et ne doivent plus les suivre en cas de migration. Néanmoins, le problème de la validité des pointeurs reste le même, sauf que dans ce cas ce n'est plus *un* thread, mais ce sont *plusieurs* threads, éventuellement distribués sur des nœuds différents, qui peuvent manipuler les pointeurs vers les données partagées. Le problème de la validité des pointeurs se pose donc maintenant dans un cadre plus général.

Nous avons choisi de résoudre ce problème en gardant la même approche *iso-adresse* : toute donnée partagée migrée ou répliquée sur plusieurs nœuds doit être toujours logée à la même adresse virtuelle. Cela a plusieurs avantages. D'une part, la conception de DSM-PM^2 est globalement plus simple, car tout défaut de page à une adresse donnée concerne toujours la même page, quel que soit le nœud où le défaut de page se produit. D'autre part, l'utilisation de la MVP n'impose aucune restriction vis-à-vis de l'utilisation des pointeurs vers les données partagées, ce qui est rarement le cas dans les MVP existantes. (DSM-Threads [75], par exemple, interdit explicitement l'utilisation de pointeurs vers les données partagées ; Millipede [48] est l'une des rares MVP qui l'autorise.) Par ailleurs, ce choix favorise aussi l'efficacité, car aucun traitement post-migration du contenu des pages ni des références à ces pages sur le nœud destinataire n'est nécessaire. Enfin, cette stratégie permet l'implémentation des protocoles basés sur la migration des threads lors des défauts de page, puisque les threads répètent automatiquement l'accès à la *même* adresse virtuelle à la sortie de la routine de gestion du défaut de page, après migration. Ceci a été illustré à travers le protocole `migrate_thread`, présenté dans la section 3.4.

```
int *ptr;

void f ()
{
  isoaddr_attr_t attr;

  isoaddr_attr_init(&attr);
  isoaddr_attr_set_status(&attr, ISO_SHARED);
  isoaddr_attr_set_protocol(&attr, hbrc_sw);
  ptr = (int *)pm2_malloc(sizeof(int), &attr);
  ...

  return;
}
```

Figure 4.8 – Allocation dynamique de mémoire partagée.

4.3.2 Allocation dynamique iso-adresse dans DSM-PM^2

La mise en place du mécanisme d'allocation dynamique dans DSM-PM^2 a nécessité l'extension de l'allocateur iso-adresse de PM^2. Cette extension a requis la prise en compte de plusieurs aspects.

Comme nous l'avons expliqué, une différence majeure concernant le comportement des données partagées vis-à-vis de la migration est que ces données ne suivent pas le thread les ayant allouées lorsque ce thread migre. En revanche, si ce thread les accède après migration, il peut produire un défaut de page qui déclenche le protocole de cohérence associé à la page. Suivant le protocole, soit le thread revient sur le nœud d'origine, soit les données sont migrées ou répliquées sur le nœud courant du thread. Les données partagées ne sont donc pas attachées à un thread et leur transfert entre les nœuds n'est pas dépendant de la migration des threads, mais du protocole de cohérence associé.

L'allocateur iso-adresse étendu doit donc gérer l'allocation de zones de mémoire avec des propriétés différentes. Nous avons distingué deux types principaux de slots alloués : des slots *privés* et des slots *partagés*. Les slots *privés* servent à stocker les piles des threads ou les données privées dynamiquement allouées par ceux-ci, qui suivent les threads en cas de migration. Un slot *partagé* n'appartient pas à un thread particulier, mais le nœud où il est alloué obtient souvent un statut privilégié, suivant le protocole de cohérence associé lors de l'allocation : nœud propriétaire pour les protocoles à écrivain unique ou nœud de référence pour les protocoles à écrivains multiples. L'utilisateur spécifie le type de mémoire désiré lors de l'appel de la primitive d'allocation. Dans l'exemple présenté sur la figure 4.8, on alloue un entier partagé. Ceci est spécifié à l'aide de l'attribut d'allocation (`ISO_SHARED`), qui permet aussi d'associer un protocole de cohérence (ici `hbrc_mw`) au bloc alloué :

```
isoaddr_attr_set_status(&attr, ISO_SHARED);
isoaddr_attr_set_protocol(&attr, hbrc_mw);
```

Pour allouer un bloc de mémoire privée censé suivre le thread appelant en cas de migration, on spécifie le statut du bloc à l'aide du même attribut. Il est inutile, dans ce cas, de spécifier un protocole de cohérence :

```
isoaddr_attr_set_status(&attr, ISO_PRIVATE);
```

Notre choix d'implémentation a été d'utiliser la même zone iso-adresse, illustrée sur la figure 4.7 pour l'ensemble des allocations dans PM^2. Au fur et a mesure des allocations, les slots disponibles (c'est-à-dire non alloués) peuvent devenir privés ou partagés.

Afin d'utiliser correctement le mécanisme d'allocation, l'utilisateur doit donc connaître a priori si les données qui seront stockées dans une zone allouée sont privées ou partagées. Il y a pourtant des cas où cette information est déterminée seulement à l'exécution et il peut être nécessaire de changer le statut d'une zone mémoire. Par exemple, dans une application Java, tout objet est a priori partagé, mais souvent certains objets ne sont utilisés que par un seul thread. Dans ce cas, il peut être intéressant, pour des raisons de performance, de considérer les objets respectifs comme étant privés au thread et de les déplacer avec le thread, en cas de migration de ce dernier, pour diminuer le coût des accès.

Pour prendre en compte ce type de situation, nous avons introduit un troisième statut de zone mémoire : `ISO_DEFAULT`. Une zone mémoire de ce type est considérée privée par défaut, mais peut devenir partagée si on détecte qu'elle est accédée par plusieurs threads. Elle est donc traitée initialement comme les slots privés du thread qui l'a allouée, qu'elle suit en cas de migration. Lorsque le système détecte que la zone mémoire est accédée par deux threads, la zone change de statut et devient partagée (`ISO_SHARED`) jusqu'à la fin du programme. Étant donné que dans le cas général les threads accèdent directement les données par des lectures et des affectations, sans passer par des primitives d'accès, le système ne peut pas détecter si une donnée est accédée par plusieurs threads du *même* nœud. En revanche, on peut détecter si elle est accédée par des threads situés sur des nœuds *différents* et c'est cet événement qui déclenche *automatiquement* le changement de statut de la zone mémoire. Par ailleurs, l'utilisateur dispose aussi d'une primitive permettant d'effectuer *explicitement* ce changement de statut. Dans le cas où DSM-PM^2 est utilisé en tant que cible d'un compilateur et que l'accès aux données partagées est contrôlé par des primitives spécifiques, l'implémentation de ces primitives peut détecter aussi les accès à une donnée partagée par plusieurs threads du *même* nœud et demander alors explicitement le changement de statut de la zone mémoire correspondante.

Pour des raisons de simplicité de la mise en œuvre, nous nous sommes limités à considérer uniquement ce type de transformation (de `ISO_DEFAULT` en `ISO_SHARED`), dont l'utilité potentielle est justifiée ci-dessus. Dans la version actuelle de DSM-PM^2, d'autres transformations n'ont pas été implémentées, car les utilisations actuelles de la plate-forme n'ont pas mis en évidence de tels besoins.

4.3.3 Choix de conception

Gestion de l'espace iso-adresse. L'extension d'`isomalloc` pour intégrer l'allocation des données partagées a posé le problème de la granularité de la gestion de l'espace iso-adresse. Cet espace est divisé en slots et les slots sont préaffectés aux différents nœuds suivant une distribution spécifiée par l'utilisateur (par blocs, cyclique, etc.). Le *nœud de rattachement* d'un slot peut y allouer des données sans en informer les autres nœuds. Lorsqu'un nœud a besoin d'allouer une zone plus large que les intervalles qui lui sont rattachés, il peut "emprunter" des slots à d'autres nœuds, grâce à une négociation, qui implique des communications. Par contre, une allocation d'un bloc de taille inférieure à la taille d'un slot reste une opération locale *quelle que soit la distribution des slots*. Dans la version initiale d'`isomalloc`, la taille

d'un slot était de 16 pages, pour qu'un slot puisse loger une pile de thread et que l'allocation de pile lors de la création d'un thread se réalise efficacement, sans négociation. Étant donné que DSM-PM^2 est une MVP à base de pages, nous avons décidé d'utiliser désormais une taille de slot d'une page. En revanche, pour qu'une pile de thread puisse toujours être logée dans une zone rattachée à un nœud unique, il est exigé que les distributions spécifiées par l'utilisateur respectent une granularité de 16 slots.

Nœud de rattachement, nœud propriétaire et nœud de référence. Remarquons que le *nœud de rattachement* spécifié par ces distributions est conceptuellement différent du *nœud propriétaire* d'une page partagée (dans le cas d'un protocole à écrivain unique) ou du *nœud de référence* (dans le cas des protocoles à écrivains multiples), qui existent uniquement pour les pages *allouées*, *partagées*. Le *nœud de rattachement* de la page correspondant à un slot est important aussi *avant* l'allocation, car c'est le nœud qui a le droit d'allouer l'intervalle d'adresses correspondant à cette page. Toute page, partagée ou privée, correspond à un intervalle d'adresses unique sur tous les nœuds et possède un nœud de rattachement unique. Lors de la migration d'un thread, les pages qui logent sa pile et ses données privées changent de nœud de rattachement.

Table d'état des pages. L'allocateur a besoin de connaître l'état des pages afin de gérer correctement les différents types d'allocations. À cet effet, nous utilisons une *table d'état des pages* sur chaque nœud. Cette table comprend une entrée pour chaque page iso-adresse, qui spécifie l'état de la page : partagée (`ISO_SHARED`), privée (`ISO_PRIVATE`), privée et partageable (`ISO_DEFAULT`) ou non allouée (`ISO_UNUSED`). La table spécifie également le nœud de rattachement de la page.

Gestion paresseuse de la table d'état. Pour des raisons d'efficacité, la table d'état des pages est gérée suivant une approche paresseuse. Une page change d'état lors des opérations d'allocation ou de libération. Afin d'éviter toute communication inter-nœuds lors de ces opérations, seule la table d'état du nœud de rattachement est mise à jour immédiatement. Les tables des autres nœuds sont actualisées par la suite : si ces nœuds ont besoin de connaître l'état d'une page, ils s'adressent au nœud de rattachement. Lorsqu'un thread s'exécutant sur un nœud N_1 accède à une page allouée dynamiquement sur un nœud distant N_2, le premier accès à cette page produit un défaut de page. Le nœud N_1 n'ayant pas d'information sur l'état courant de la page, une requête est adressée préalablement au nœud de rattachement N_2. La réponse à cette requête permet de mettre à jour la table d'état du nœud N_1. Ces communications sont pourtant rares : un nœud adressera une requête d'information sur l'état d'une page au plus une fois, car soit le défaut de page concerne une page partagée (ou partageable), qui restera partagée jusqu'à la fin de l'application, soit il s'agit d'un défaut de page ordinaire, issu d'une erreur de programmation, et alors l'application s'arrête.

4.3.4 Vue détaillée du traitement d'un défaut de page dans DSM-PM^2.

Le déroulement du traitement d'un défaut de page dans DSM-PM^2 est détaillé sur le dans le schéma présenté sur la figure 4.9. Le support générique de DSM-PM^2 capte le défaut de page et détermine le traitement à effectuer de la manière suivante.

1. On teste si la page concernée par l'accès est une page partagée statique. Si c'est le cas, on consulte la table des pages partagées pour déterminer le protocole associé à la page, ensuite on appelle la routine de gestion du défaut de page (en lecture ou en écriture, suivant le type d'accès détecté) du protocole respectif.

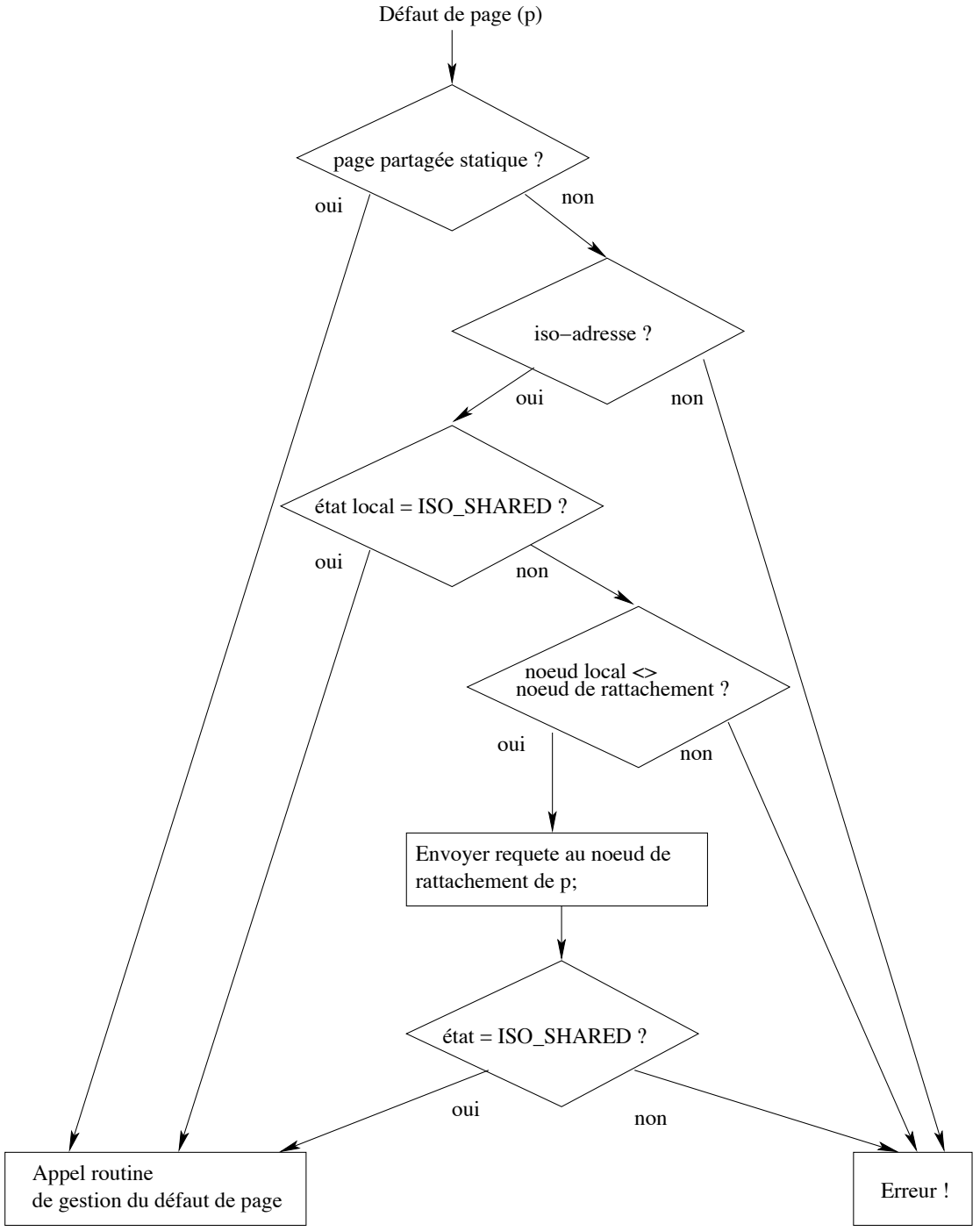


FIG. 4.9 – Zoom sur le traitement d'un défaut de page dans DSM-PM^2

2. Sinon, on teste si la page appartient à la zone d'allocation dynamique iso-adresse. Si ce n'est pas le cas, il s'agit d'un défaut de page causé par une erreur de programmation.

3. Si c'est une page iso-adresse, on consulte la table d'état locale pour tester si la page est

connue comme partagée. Si c'est le cas, on appelle la routine de gestion du défaut de page correspondant au protocole associé à la page (en tenant compte du type d'accès).

4. Sinon, un autre test est effectué : si le nœud de rattachement de la page est le nœud local, alors l'état de la page ne peut être que `ISO_UNUSED`, donc il s'agit d'une erreur de programmation, puisqu'on accède à une zone non allouée. En effet, l'état ne peut être ni `ISO_PRIVATE`, ni `ISO_DEFAULT`, car dans ce cas la page serait allouée et accessible en lecture et en écriture, donc le défaut de page n'aurait pas pu se produire.
5. Enfin, il reste le cas où le nœud de rattachement est un nœud distant, alors que l'état indiqué par la table locale n'est pas `ISO_SHARED`. Dans cette situation, il est nécessaire d'adresser une requête au nœud de rattachement, pour demander l'état de la page. Plusieurs cas peuvent se produire, suivant l'état spécifié par la table d'état des pages du nœud de rattachement.

   (a) Si la page est partagée (`ISO_SHARED`), le nœud de rattachement répondra à la requête en indiquant cet état, ainsi que le protocole de cohérence associé à la page et le propriétaire courant (ou le nœud de référence). Cette réponse déclenche la mise à jour de la table d'état des pages sur le nœud demandeur ainsi que la création et l'initialisation d'une nouvelle entrée dans la table des pages partagées. Ensuite, suivant le type d'accès, la routine de gestion du protocole associé à la page est appelée.

   (b) Si la page est privée mais partageable (`ISO_DEFAULT`), la réception de la requête d'information sur le nœud de rattachement est interprétée comme un accès à la page sur un nœud différent. La page devient alors partagée, ce qui déclenche l'actualisation de quelques structures de données sur le nœud de rattachement. La page est détachée de la liste des pages du thread, son état devient `ISO_SHARED` dans la table d'état des pages et une entrée lui est associée dans la table des pages partagée. Le protocole associé à la page sera le protocole spécifié par l'utilisateur lors de l'allocation (cf. exemple 4.8). Si aucun protocole n'a été spécifié, le protocole par défaut sera utilisé. Une fois ces opérations effectuées, le nœud de rattachement renvoie les informations sur le *nouvel* état de la page au nœud demandeur, en déclenchant les mêmes actions que dans le cas précédent.

   (c) Enfin, si l'état de la page est `ISO_PRIVATE` ou `ISO_UNUSED`, cette information est transmise au nœud demandeur. L'application est alors arrêtée, car l'accès révèle une erreur de programmation.

4.4 Validation préliminaire : le problème du voyageur de commerce

4.4.1 Présentation de l'application

Afin d'illustrer le fonctionnement de DSM-PM^2 en tant que plate-forme expérimentale de comparaison de protocoles, nous avons réalisé une première expérience consistant à exécuter un programme qui résout le problème du voyageur de commerce pour 17 villes placées aléatoirement. Le programme explore l'arbre des possibilités en s'arrêtant dès que le chemin qu'il explore est plus long que le plus court chemin courant. La valeur du plus court chemin courant est la seule variable partagée du programme accédée en lecture *et* en écriture.

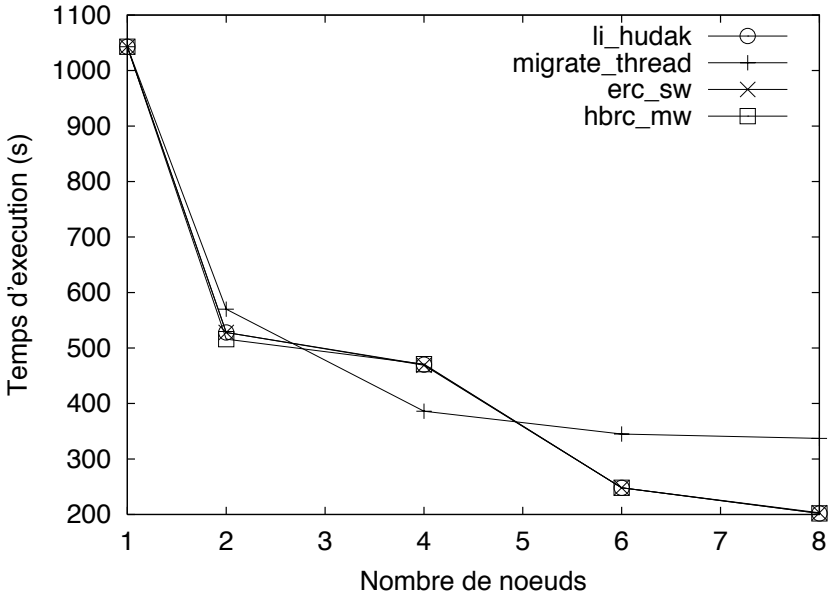


FIG. 4.10 – Résolution du problème du voyageur de commerce sur BIP/Myrinet.

L'objectif premier de cette expérience a été de comparer les performances des protocoles basés sur le transfert des pages à celles des protocoles basés sur la migration de threads. Le code de l'application est strictement identique à chaque mesure, sans qu'une recompilation soit nécessaire. Le protocole de cohérence à utiliser est un paramètre à fournir au programme lors de son lancement.

Dans ces tests préliminaires, chaque nœud exécute un thread de l'application.

4.4.2 Comparaison des protocoles : transfert de pages *vs* migration de thread

Les figures 4.10 et 4.11 présentent les temps d'exécution pour 4 protocoles intégrés qui implémentent soit la cohérence séquentielle, soit la cohérence à la libération. Les mesures ont été effectuées sur une grappe BIP/Myrinet à nœuds PPro à 200 MHz et sur une grappe SCI à nœuds PII à 450 MHz.

Étant donné que tous les accès à l'unique variable partagée sont toujours protégés par des verrous et qu'il n'y a pas de faux partage, les avantages de la cohérence à la libération par rapport à la cohérence séquentielle ne sont pas mis en évidence dans cet exemple. Mais nous remarquerons que tous les protocoles basés sur le transfert des pages produisent de meilleures performances que le protocole basé sur la migration de thread *dans le cas de cette application*. Ceci a une explication simple : tous les threads de l'application migrent vers le nœud qui détient la variable partagée, qui devient ainsi surchargé. Comme nous l'avons expliqué dans la section précédente, l'efficacité d'une stratégie basée sur la migration des threads dépend de la distribution des données partagées et des schémas d'accès à ces données. Dans notre cas, un déséquilibre est créé par le fait que la donnée partagée se réduit à un entier unique accédé par tous les threads de l'application. Nous pouvons prévoir un meilleur comportement du protocole `migrate_thread` pour des applications dans lesquelles les données sont distribuées et accédées uniformément sur les différents nœuds.

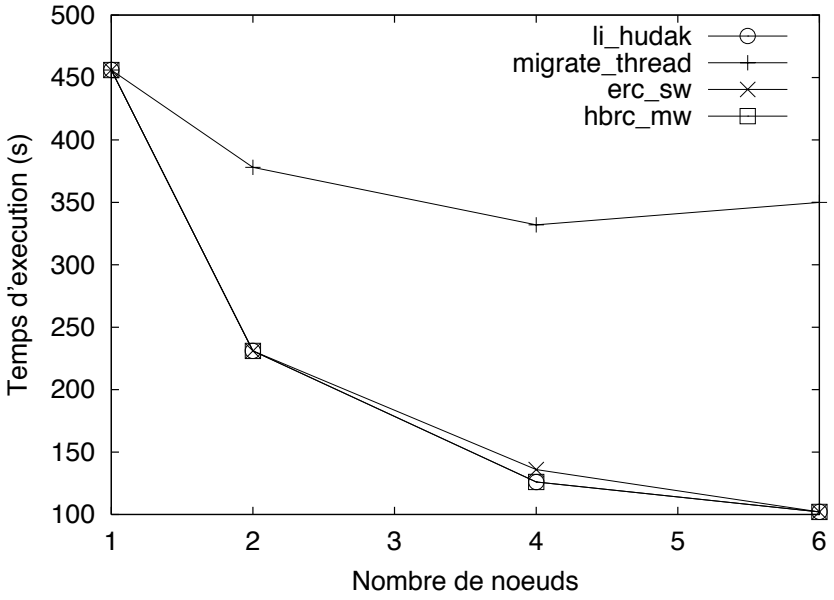


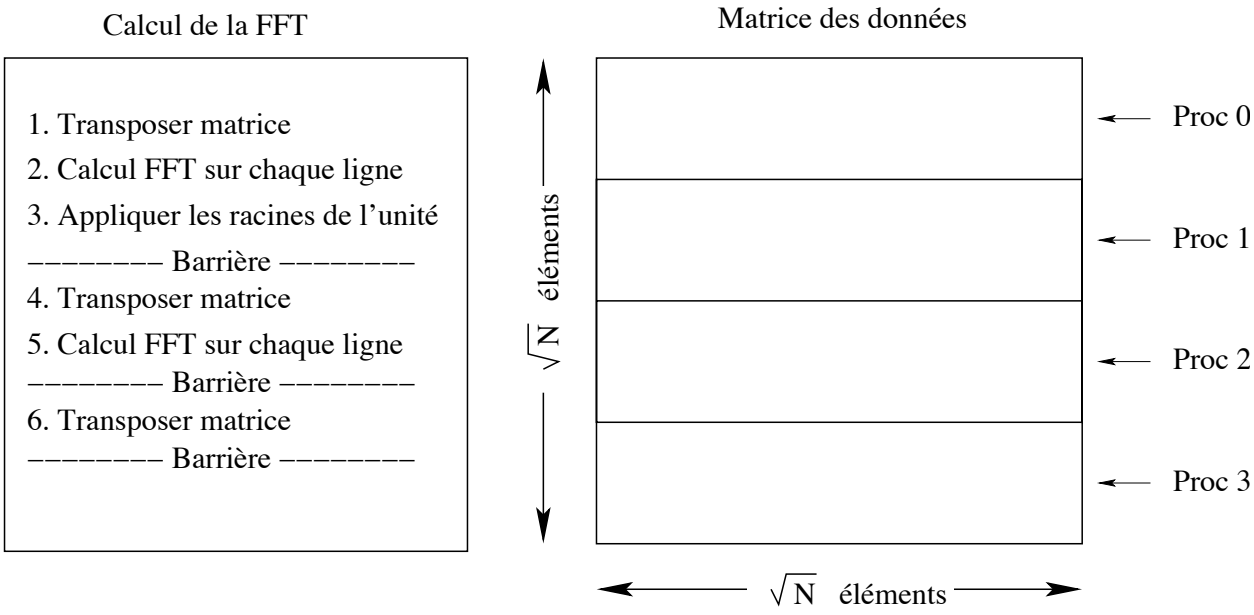
FIG. 4.11 – Résolution du problème du voyageur de commerce sur SISCI/SCI.

4.5 Une application : calcul de la transformée de Fourier

4.5.1 Présentation de l'application

Afin d'évaluer l'efficacité de nos deux protocoles de cohérence à la libération par rapport à des protocoles de cohérence séquentielle, nous avons utilisé comme application de test un noyau FFT 1D (*Fast Fourier Transform*, version unidimensionnelle) extrait des benchmarks SPLASH-2 [102], que nous avons adapté pour une exécution multithread au-dessus de DSM-PM^2. Cette application a été choisie essentiellement pour la facilité de son portage. Les données de l'application sont constituées d'une matrice $\sqrt{n} \times \sqrt{n}$ contenant N points à transformer et d'une autre matrice $\sqrt{n} \times \sqrt{n}$ contenant N racines complexes du nombre réel 1. Une troisième matrice auxiliaire de la même taille est utilisée pour stocker des valeurs intermédiaires. La matrice des points et la matrice auxiliaire sont lues et écrites pendant l'exécution du programme, alors que la matrice des racines de l'unité est en lecture seule, une fois initialisée. Les trois matrices sont allouées sur les processeurs par blocs de lignes, de manière à ce que chaque processeur possède un ensemble de lignes contiguës, comme nous pouvons le voir sur la figure 4.12.

Le principe de l'algorithme utilisé pour le calcul de la transformée de Fourier est présenté sur la figure 4.12. Dans une première étape, la matrice source est transposée dans la matrice auxiliaire. Dans une deuxième étape, la FFT 1D est calculée pour chaque ligne de la matrice auxiliaire, les résultats étant stockés toujours dans la matrice auxiliaire. Dans la troisième étape, les éléments correspondants de la matrice des racines de l'unité sont appliqués à chaque élément de la matrice auxiliaire. Dans la quatrième étape, la matrice auxiliaire est transposée dans la matrice source. À l'étape 5, la FFT 1D est calculée pour chaque ligne de la matrice source, les résultats étant stockés toujours dans la matrice source. Enfin, à l'étape 6, la matrice source est transposée de nouveau dans la matrice auxiliaire. Les communications ont lieu lors de trois étapes de transposition et



(a) Algorithme (b) Distribution des données

FIG. 4.12 – Principe de l'algorithme de calcul de la FFT et schéma de distribution des données.

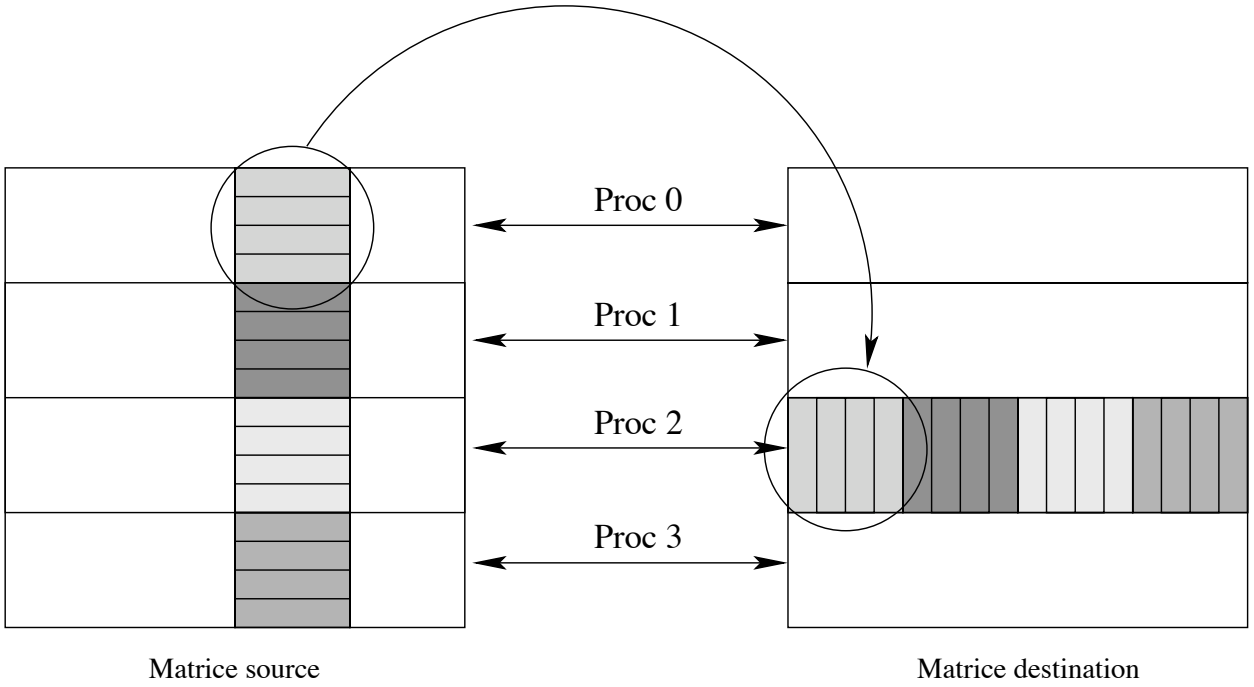


FIG. 4.13 – Détail sur la phase de transposition dans l'algorithme de la FFT.

Opération	TCP/Fast Ethernet (PPro 200 MHz)	SISCI/SCI (PII 450 MHz)
Défaut de page	23	11
Transmission requête	370	38
Transfert de page	3900	119
Protection d'une page	22	12

TAB. 4.4 – Coût des opérations élémentaires (μs).

nécessitent des communications *all-to-all*. Les transpositions se font par blocs : le processeur i transpose une sous-matrice du processeur $i+1$, ensuite une sous-matrice du processeur $i+2$, etc.

Les tests ont été effectués sur deux plates-formes différentes : 1) une grappe de PC Pentium Pro 200 MHz sous Linux 2.2.13 interconnectés par un réseau lent (Fast-Ethernet sous TCP) ; ces mesures n'ont pas pu être exécutées sur la grappe de PC PII 450 MHz reliés par Ethernet, pour des raisons d'indisponibilité de la plate-forme ; 2) une grappe de PC Pentium II 450 MHz interconnectés par un réseau rapide (SCI géré par l'interface SISCI). Les coûts des opérations principales utilisées par les protocoles sont donnés dans le tableau 4.4 pour chaque plate-forme utilisée. Nous avons mesuré le temps total d'exécution de l'application pour 3 protocoles : `erc_sw` et `hbrc_mw` pour la cohérence à la libération et `li_hudak`, qui sert de référence pour la cohérence séquentielle. Nous nous sommes ici limité à des tests avec des protocoles basés sur le transfert des pages.

4.5.2 Comparaison des protocoles : cohérence à la libération *vs* cohérence séquentielle

Dans les figures 4.14 et 4.15, nous comparons les deux protocoles de cohérence à la libération au protocole de cohérence séquentielle `li_hudak` sur une configuration à 4 nœuds, en utilisant un thread utilisateur par nœud, respectivement sur les grappes utilisant Fast Ethernet et SCI. Les temps sur SCI sont meilleurs que ceux sur Fast Ethernet d'un facteur 20 environ, à cause des performances relatives des deux réseaux en débit et en latence, mais également à cause des performances différentes des processeurs utilisés. Dans les deux cas, les protocoles `erc_sw` et `hbrc_mw` permettent une exécution beaucoup plus efficaces que le protocole `li_hudak`. En effet, l'intérêt des protocoles de cohérence à la libération par rapport à ceux de cohérence séquentielle se manifeste particulièrement en présence de faux partages de type lecture/écriture, ce qui est précisément le cas ici. Les mesures sur TCP illustrent la supériorité du protocole `hbrc_mw` par rapport à `erc_sw`, grâce aux d'écrivains multiples. En effet, le nombre de pages transférées est plus faible pour `hbrc_mw` que pour `erc_sw`, et cet avantage est d'autant plus sensible que le réseau est peu performant par rapport au processeur.

4.5.3 Influence du multithreading

Nous avons également étudié l'influence du multithreading sur l'efficacité des différents protocoles, en faisant varier le nombre de threads par nœud de l'application pour une même taille du problème (256 ko) et pour un nombre fixé de nœuds (4 nœuds). Les temps de calcul observés sont présentés dans les tableaux 4.5 et 4.6. Les temps obtenus pour TCP mettent évidence le gain en efficacité que l'on peut obtenir grâce aux recouvrements des temps d'attente des messages du réseau par du calcul lorsqu'on utilise plusieurs threads de l'application par nœud. Dans

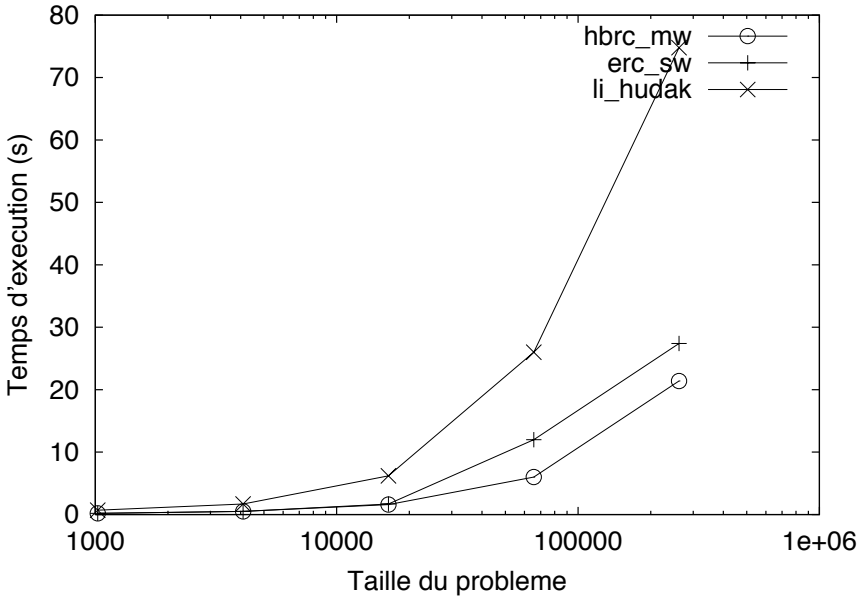


FIG. 4.14 – FFT sur TCP/Fast Ethernet (4 nœuds).

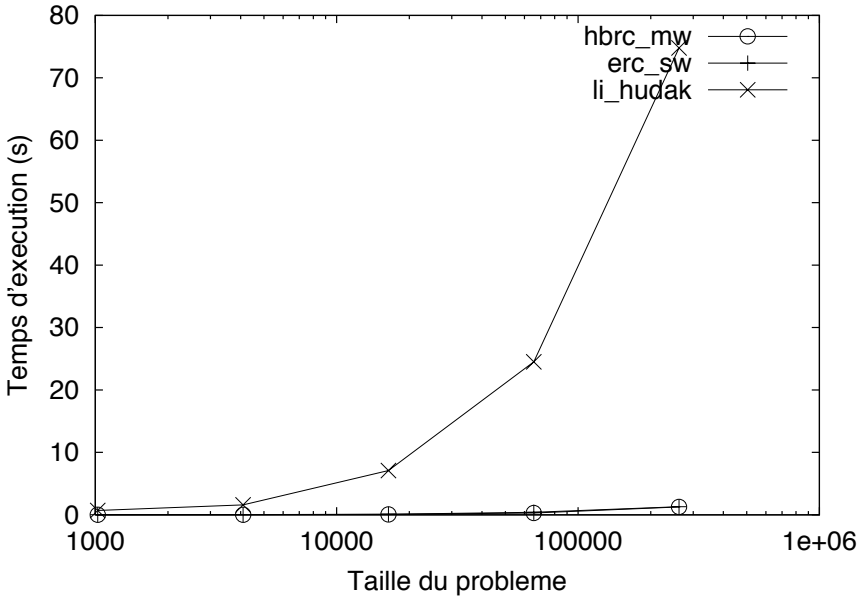


FIG. 4.15 – FFT sur SISCI/SCI (4 nœuds).

Threads/nœud	1	2	4	8
li_hudak	74.8	54.8	54.4	57.5
erc_sw	27.2	21.0	19.6	24.1
hbrc_mw	21.4	20.9	20.9	21.1

TAB. 4.5 – FFT sur TCP/Ethernet (s).

Threads/nœud	1	2	4	8
li_hudak	79	57	44	36
erc_sw	1.3	1.3	1.4	1.4
hbrc_mw	1.2	1.2	1.2	1.2

TAB. 4.6 – FFT sur SISCI/SCI (s).

cet exemple, les meilleurs temps d'exécution sont obtenus pour une ratio de 4 threads/nœud. Au delà, le surcoût du multithreading commence à compenser le gain dû au recouvrement.

Ces phénomènes ne sont pas visibles sur la grappe SCI, sur laquelle l'effet de recouvrement n'est pratiquement pas visible, car le réseau est très rapide. Nous remarquerons néanmoins que le surcoût du multithreading n'est pas significatif. Le multithreading est donc supporté par la plate-forme pour un coût négligeable.

4.6 Conclusion

Ce chapitre a été consacré à l'implémentation et à l'évaluation des performances de la plate-forme DSM-PM^2. Nous avons présenté deux problèmes importants. Le premier concerne la prise en compte du multithreading dans les protocoles de cohérence. Afin d'assurer une exécution plus efficace grâce aux recouvrements potentiels des temps d'attente par des calcul, nous avons dû gérer un certain nombre de situations de concurrence dans l'exécution des actions de cohérence. Nous avons détaillé quelques-unes de ces situations. Nous avons ensuite présenté quelques mesures de coût pour les opérations de base de la plate-forme. Nous avons évalué le coût de deux stratégies de base (la migration des threads et le transfert des pages) et nous avons mis en évidence l'importance de critères secondaires tels que la distribution des données pour les performances globales des applications.

Une partie important du chapitre a été consacrée à la description détaillée de l'extension que nous avons opérée sur l'allocateur dynamique de mémoire de PM^2, afin de permettre l'allocation des données partagées. Les principaux avantages de nos choix de conception sont expliqués : nous utilisons une approche iso-adresse, permettant de garantir la validité des pointeurs sur les données partagées, tout en gardant un fonctionnement simple et efficace lors des transferts des pages. L'approche permet également l'implémentation des protocoles basés sur la migration des threads lors des défauts de page, comme nous l'avons illustré à travers le protocole `migrate_thread`, introduit dans la section 3.4.

Enfin, nous avons présenté quelques mesures préliminaires qui évaluent les performances relatives des protocoles intégrés. Nous avons également donné une évaluation préliminaire de l'influence de l'utilisation du multithreading au niveau applicatif sur les performances de l'application.

Dans le chapitre suivant, nous présenterons d'autres mesures de performances, pour 5 applications Java compilées avec le système Hyperion et exécutées au-dessus de DSM-PM^2.

Chapitre 5

Application : DSM-PM^2 - cible d'un compilateur Java

Afin de valider la plate-forme DSM-PM^2 en tant que cible d'un système de compilation d'un langage basé sur le paradigme de la mémoire partagée, nous avons intégré DSM-PM^2 dans un environnement d'exécution Java appelé Hyperion, développé par le groupe de Philip Hatcher, à University of New Hampshire (NH, USA). Nous avons établi une collaboration active avec Philip Hatcher lors des phases de conception et d'implémentation de la version courante de l'exécutif d'Hyperion, notamment en ce qui concerne la gestion de la mémoire partagée. L'objectif du projet Hyperion est de permettre l'exécution transparente d'applications multithreads Java sur grappes de PC. Pour supporter l'abstraction d'une mémoire partagée accessible par tous les threads de l'application, indépendamment de leur localisation, Hyperion utilise DSM-PM^2 à travers deux protocoles de cohérence spécifiquement conçus à partir de la spécification du modèle de mémoire de Java [38]. Ce chapitre présente l'approche choisie par Hyperion et décrit les détails d'implémentation de ce système au-dessus de DSM-PM^2. La section 5.1 donne un aperçu général d'Hyperion et des approches similaires. L'architecture du système est présentée dans la section 5.2. Ensuite, la section 5.3 discute les détails de l'implémentation du modèle de mémoire de Java sur DSM-PM^2 par deux protocoles de cohérence différents. Enfin, dans la section 5.4 nous présentons les performances relatives des deux protocoles suite à des expérimentations avec 5 applications à caractéristiques différentes exécutées sur deux types de grappes.

5.1 Exécution transparente des threads Java sur grappes de PC

Le projet Hyperion est fondé sur deux constats. Premièrement, le langage Java est devenu de plus en plus utilisé pour la programmation parallèle, ce qui s'explique par ses deux importantes propriétés : d'une part, il a été conçu pour être portable, d'autre part, il est explicitement parallèle, grâce au modèle de programmation multithread. Deuxièmement, on peut constater que les grappes de PC représentent les machines parallèles typiques d'aujourd'hui, essentiellement à cause de leur très bon rapport performance/prix. Dans ce contexte, l'objectif de fournir un environnement permettant l'exécution efficace de programmes Java sur grappes de PC devient particulièrement intéressant.

L'exécution distribuée d'applications Java sur grappes a été abordée essentiellement de deux

manières selon la façon dont le programmeur "voit" la grappe. Dans la première approche, les différents nœuds de la grappe sont pris en compte au niveau de la programmation. Le programmeur doit donc gérer explicitement les communications inter-nœuds, en utilisant des invocations de méthodes à distance (`RMI`, *remote method invocations*) ou des bibliothèques se passage de messages pour communiquer entre plusieurs machines virtuelles Java. À l'opposé, dans la deuxième approche, le programmeur "voit" une seule machine virtuelle Java et les nœuds de la grappe sont "cachés" par l'implémentation.

Les deux approches ont été illustrées dans des environnements permettant l'exécution distribuée d'applications Java sur grappes.

- Quelques environnements, tels que Java/HPC++ [17], Proactive [18] ou Manta [67] ont choisi de rendre visibles au programmeur les nœuds de la grappe et de gérer les communications explicitement à l'aide du mécanisme `RMI`. D'autres, tels que JPVM [28] et HPJava [35] ont choisi une approche similaire, mais la gestion des communications est basée sur des bibliothèques de communication.
- Un pas vers la transparence est fait par des environnements comme JavaParty [84] et Do ! [58], qui rajoutent une couche dont le rôle est de cacher des détails liés à la distribution et à la localité. Néanmoins, les objets distants doivent être déclarés explicitement dans le programme par des annotations.
- Enfin, d'autres systèmes implémentent des interprètes distribués qui offrent au programmeur l'image d'une machine virtuelle Java unique et projettent le tas de cette machine sur une MVP existante. C'est l'approche choisie par MultiJav [21] et Java/DSM [103]. Cette dernière utilise TreadMarks pour supporter l'abstraction de la mémoire virtuellement partagée. Le choix d'une telle MVP disponible en tant que système fermé, non configurable, présente un certain nombre de limites en ce qui concerne la performance. D'une part, aucune MVP de ce type ne fournit, à notre connaissance, un support spécifique pour la cohérence Java. Le modèle de cohérence le plus souvent utilisé dans ce cas est celui de la cohérence à la libération, ce qui imposent certaines limitations à l'utilisation[1]. Ensuite, un tel système ne permet pas de tirer profit des optimisations potentielles spécifiques à ce modèle de cohérence. Une approche plus adaptée nous semble être celle du projet cJVM [8], qui implémente une machine virtuelle Java distribuée complète et gère explicitement la cohérence Java.

L'approche d'Hyperion est proche de celle des derniers systèmes présentés. L'objectif est celui de la transparence totale : les programmeurs doivent pouvoir exécuter des applications multithread Java existantes sur des grappes, *sans modifications*. Mais, au lieu d'utiliser une approche basée sur l' *interprétation*, Hyperion choisit une approche de type *compilation* : le *bytecode* Java est compilé en code C. Les aspects liés à l'implémentation des threads et des communications restent à la charge d'un environnement multithread distribué tel que PM^2 et l'implémentation du modèle de mémoire de Java repose sur la plate-forme DSM-PM^2.

Le système Hyperion a été conçu pour une utilisation adaptée au développement d'applications de calcul à hautes performances. Le programmeur développe son application sur sa station de travail locale et ensuite soumet l'application à un serveur distant, en vue d'une exécution distribuée. Le paradigme traditionnel qui consiste à charger du *bytecode* depuis un serveur distant et à l'exécuter localement est inversé : le programmeur envoie le bytecode au serveur distant, pour une exécution efficace. À l'arrivée sur ce serveur, le bytecode est traduit en code natif. Ce pro-

[1] Il a été démontré que la cohérence Java est équivalente à la cohérence à la libération sous certaines conditions. Voir [37] pour une discussion détaillée.

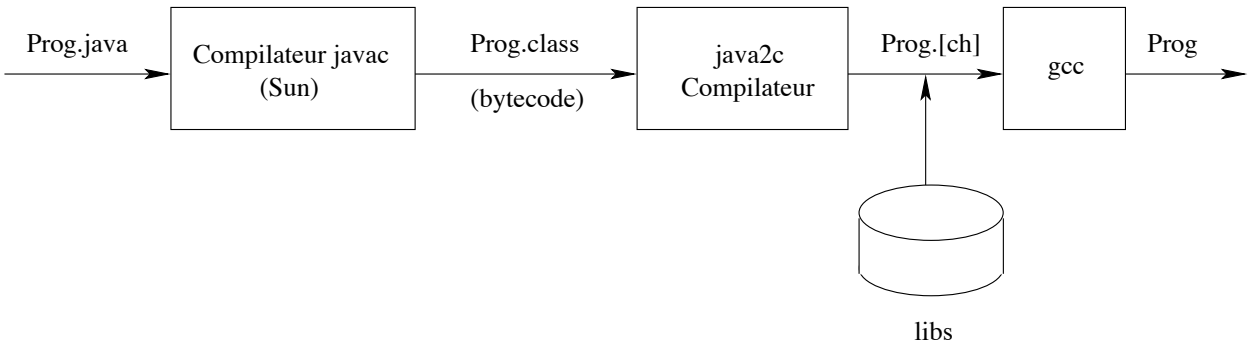


FIG. 5.1 – Compilation de programmes Java avec Hyperion

cessus utilise un compilateur ad-hoc de code Java (appelé `java2c`) qui produit du code C et un compilateur C standard tel que `gcc`.

5.2 Le système de compilation Hyperion

5.2.1 Architecture

La chaîne de compilation d'Hyperion est représentée sur la figure 5.1. Les classes de l'application sont d'abord compilées par le compilateur `java2c` intégré dans Hyperion. Le code C généré est compilé par le compilateur `gcc` et lié avec l'exécutif d'Hyperion et avec les autres bibliothèques externes.

La génération de code dans `java2c` suit un schéma simple. Chaque instruction virtuelle est transformée en une instruction C séparée, suivant une approche similaire à celle des compilateurs Harissa [79] ou Toba [86]. Par conséquent, des optimisations telles que l'élimination des variables auxiliaires créées pendant ce processus restent à la charge du compilateur C.

Les instructions plus complexes de la machine virtuelle, telles que les accès aux objets, sont compilées en appels à des macros. Ce schéma a plusieurs avantages. Tout d'abord, la génération de code par le compilateur se trouve simplifiée. Ensuite, l'approche permet la réalisation d'expérimentations avec ces macros, pour des études de performances. Enfin, ce schéma comporte des avantages concernant la portabilité. Par exemple, le système pourrait être porté facilement sur un environnement à mémoire *physiquement* partagée en adaptant les macros concernant la mémoire virtuellement partagée.

Le compilateur `java2c` comprend également un optimiseur qui améliore l'efficacité des accès à la mémoire virtuellement partagée. Par exemple, si un objet est référencé au sein d'une boucle, l'optimiseur place à l'extérieur de la boucle le code permettant d'obtenir une copie locale de l'objet. À l'intérieur de la boucle, l'objet peut alors être accédé efficacement par l'intermédiaire d'un simple pointeur. Ce type d'optimisation doit être supporté à la fois par une analyse lors de la phase de compilation ainsi qu'au niveau de l'exécutif, pour s'assurer que le cache local n'est pas vidé pendant l'exécution de la boucle.

L'exécutif d'Hyperion est constitué de plusieurs modules qui interagissent les uns avec les autres (figure 5.2). Voici une brève description des plus importants.

`loadIntoCache`	Charger un objet depuis la mémoire principale dans le cache local du thread
`invalidateCache`	Invalider tous les objets du cache local
`updateMainMemory`	Mettre à jour la mémoire principale avec les modifications apportées aux objets du cache local
`get`	Lire la valeur d'un champ d'un objet chargé préalablement dans le cache
`put`	Modifier la valeur d'un champ d'un objet chargé préalablement dans le cache

TAB. 5.1 – Interface entre le module de gestion de la mémoire d'Hyperion et la couche d'implémentation.

Support de l'interface de programmation Java. La version courante d'Hyperion repose sur le JDK 1.1 de Sun Microsystems pour le support de l'interface de programmation d'applications (API) de Java. Toutes les classes de l'API qui ne comportent pas de méthodes natives peuvent être compilées par `java2c`. Les classes qui comportent des méthodes natives doivent être récrites pour Hyperion. Malheureusement, le JDK 1.1 contient un grand nombre de méthodes natives éparpillées dans les classes de l'API. À présent, seulement quelques-unes de ces méthodes ont été implémentées dans Hyperion et le support de l'API est par conséquent limité.

La gestion des threads. Le module de gestion des threads fournit le support pour l'implémentation des threads Java. Ce support inclut des opérations de création/destruction et de synchronisation par des verrous d'exclusion mutuelle. Pour des raisons de portabilité, l'interface de ce module a été conçue pour correspondre aux principales fonctions de manipulation des threads spécifiées par POSIX. De plus, le module fournit une API pour la migration des threads.

La gestion des communications. Le module de gestion des communications supporte la transmission de messages entre les nœuds de la grappe sous-jacente. L'interface de programmation repose sur l'invocation asynchrone de services du côté du récepteur. Ce type d'interface est obligatoire, car la plupart des communications, avec ou sans acquittement, doivent se dérouler sans la participation explicite du nœud distant : les requêtes entrantes sont gérées par un thread démon qui s'exécute en parallèle avec les threads de l'application. Par exemple, dans notre implémentation du modèle de mémoire de Java, tout nœud de la grappe peut demander des données à un autre nœud de manière asynchrone.

La gestion de la mémoire. Le module de gestion de la mémoire est responsable de l'implémentation du modèle de mémoire de Java. Son interface de programmation repose sur la spécification opérationnelle du modèle de mémoire de Java [38]. Ce modèle utilise les notions de *mémoire principale* et de *caches locaux* des threads. Les primitives de base du module sont énumérées dans la table 5.1. Ces primitives manipulent la mémoire principale et les caches des threads. L'implémentation du modèle de mémoire de Java dans Hyperion est discutée en détail dans la section 5.3.1.

Le module de gestion de la mémoire comporte également des mécanismes d'allocation dynamique d'objets, de synchronisation distribuée et un ramasse-miettes. Les *moniteurs* de Java et les

méthodes associées (`wait`/`notify`) sont supportés en associant des verrous d'exclusion mutuelle et des variables de condition fournis par le module des threads à des objets gérés par le module de gestion de la mémoire d'Hyperion.

Il faut souligner que toute la conception du module de gestion de la mémoire d'Hyperion et l'interface de programmation correspondante reposent sur la notion d'objet. Par contre, le support sous-jacent de mémoire virtuellement partagée fourni par DSM-PM^2 est à base de pages. L'interface de programmation du module de gestion de la mémoire masque ce détail d'implémentation vis-à-vis du reste du système. Plus de détails sont donnés dans la section 5.3.

Le module d'équilibrage de charge. Le module d'équilibrage de charge est responsable du choix du nœud où sera placé tout thread nouvellement créé. La stratégie courante est simple : les threads sont affectés aux nœuds de manière cyclique. Hyperion utilise un algorithme distribué dans lequel chaque nœud place cycliquement les threads qu'il crée, indépendamment des autres nœuds.

5.2.2 Implémentation

Chaque module de haut niveau de l'exécutif d'Hyperion, et en particulier les modules de gestion des threads, des communications et de la mémoire, exige un certain nombre de fonctionnalités supportées par la couche d'implémentation, de plus bas niveau. L'implémentation courante de l'exécutif d'Hyperion repose sur l'environnement multithread distribué PM^2, qui fournit les fonctionnalités requises par tous les modules. Néanmoins, d'autres implémentations sont possibles : par exemple, une autre implémentation du module de mémoire, à base d'objets, est en cours de développement. Nous nous intéresserons uniquement à l'implémentation basée sur PM^2, la seule qui soit fonctionnelle aujourd'hui.

L'interface de programmation de PM^2 permet la création des threads, localement ou à distance, et la communication inter-nœud via le mécanisme d'appel de procédure à distance (RPC). De plus, PM^2 fournit la migration préemptive et transparente des threads, qui peut être utilisée pour implémenter des stratégies d'équilibrage dynamique de charge. Enfin, l'accès à un espace d'adressage global est offert par DSM-PM^2, qui fournit des mécanismes permettant la mise en œuvre aisée de protocoles de cohérence.

La plupart des primitives fournies par les modules de gestion des threads, des communications et de la mémoire sont projetées directement sur les primitives correspondantes de PM^2 et de DSM-PM^2.

Implémentation du module de gestion des threads. Le module de gestion des threads est une fine couche d'interfaçage avec Marcel, la bibliothèque des threads de PM^2, qui offre une interface de type POSIX permettant la manipulation efficace des threads en espace utilisateur. La plupart des fonctions de l'API de Marcel proposent la même interface de programmation que les fonctions POSIX Threads correspondantes. Il est tout de même important de noter que le module de gestion des threads d'Hyperion n'utilise pas directement l'interface de programmation de Marcel, comme il serait typique dans le cas d'utilisation d'une bibliothèque de threads POSIX classique. Il utilise une interface de plus haut niveau fournie par PM^2, qui met en œuvre une intégration fine des threads et des communications.

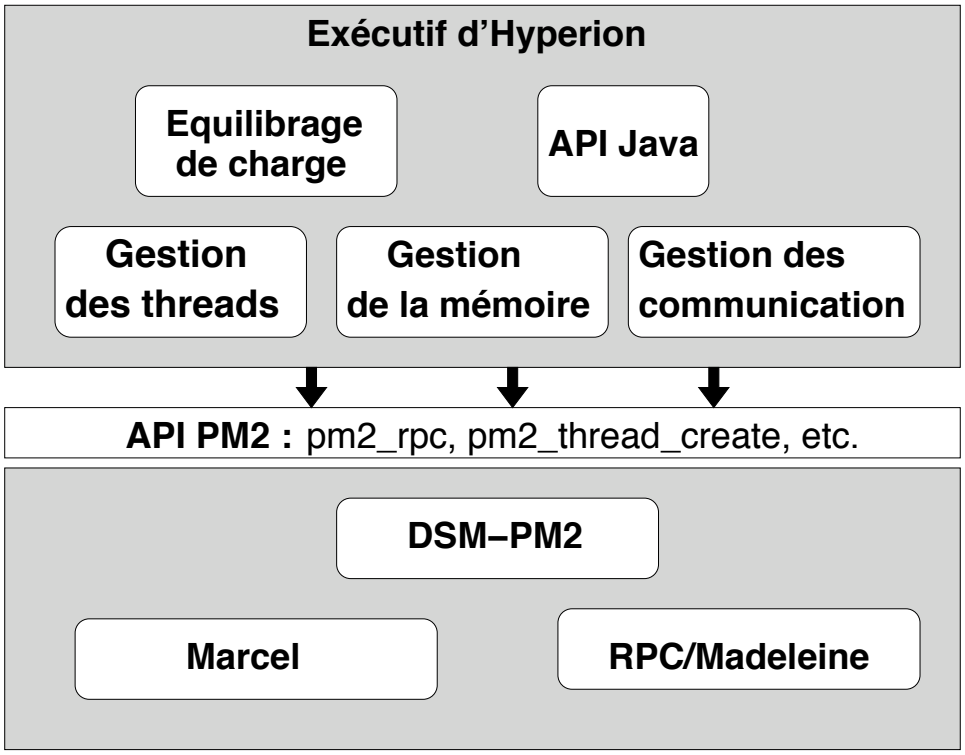


FIG. 5.2 – Architecture de l'exécutif d'Hyperion

Implémentation du module de gestion des communications. Le module de gestion des communications est implémenté à l'aides des appels de procédure à distance de PM^2 (RPC), ce qui permet l'invocation de services à distance. Sur le nœud distant, les invocations de RPC peuvent être prises en charge soit par un thread existant, soit par un nouveau thread. Cette dernière fonctionnalité a permis une implémentation aisée des primitives du module de communication d'Hyperion. Cette implémentation basée sur des les RPC de PM^2 présente des avantages importants concernant la portabilité, car elle permet à Hyperion d'être fonctionnel au-dessus d'un grand nombre de grappes basées sur différentes interfaces de communications comme SISCI, BIP, VIA, MPI, TCP.

Implémentation du module de gestion de la mémoire. Les primitive de gestion de la mémoire présentées dans la table 5.1 sont implémentées en utilisant l'interface de programmation de DSM-PM^2. Ceci représente une utilisation typique de la plate-forme DSM-PM^2, qui fournit les mécanismes de base pour l'implémentation de protocoles de cohérence multithreads et pour leur mise au point, suivant les besoins des couches supérieures. Dans ce cadre, nous avons implémenté le modèle de cohérence de Java à l'aide de deux protocoles de cohérence que nous décrivons plus loin. La conception de ces protocoles a été faite en tenant compte des hypothèses et des contraintes liées à leur utilisation dans Hyperion. Il s'agit d'une approche de type *co-design*, qui tire profit de la flexibilité de DSM-PM^2 et de la richesse des possibilités d'interfaçage des deux systèmes.

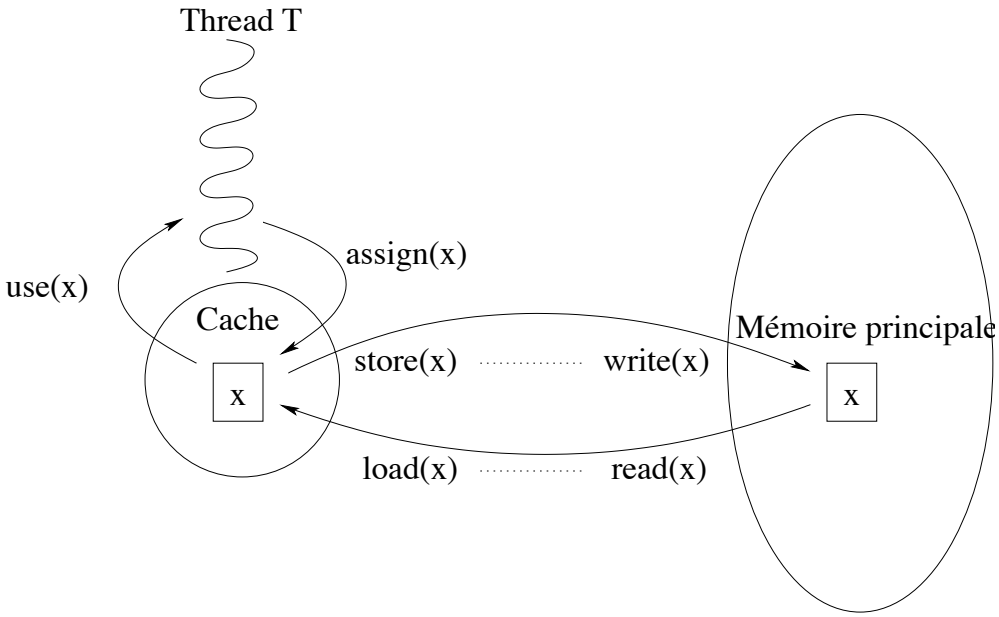


FIG. 5.3 – Interactions entre le cache d'un thread et la mémoire principale dans le modèle de mémoire de Java.

5.3 Implémentation du modèle de cohérence de Java

5.3.1 Le modèle de mémoire de Java

Un aspect central dans la conception d'Hyperion concerne l'implémentation du modèle de mémoire de Java. Hyperion doit fournir l'illusion d'une mémoire partagée, uniformément accessible, indépendamment de la localisation des objets sur les différents nœuds de la grappe.

La modèle de mémoire de Java permet aux threads d'utiliser leur mémoire privée en guise de cache pour stocker des valeurs chargées depuis la mémoire principale. L'utilisation des caches produit une amélioration importante de l'efficacité si l'application est caractérisée par un haut de degré de localité temporelle et accède plusieurs fois un objet du cache avant que le cache soit invalidé. La spécification du langage Java [38, Chapitre 17] décrit de manière très détaillée comment un thread peut utiliser son cache. Il spécifie que chaque thread possède conceptuellement un *cache local*, qui interagit avec une *mémoire principale* commune (Figure 5.3). Un thread exécute des actions de type *use* et *assign* pour accéder des variables du cache local. L'action *use* lit une variable du cache en vue d'une utilisation par le thread. L'action *assign* écrit dans une variable présente dans le cache une nouvelle valeur produite par le thread.

Le transfert de la valeur d'une variable depuis la mémoire principale vers le cache du thread correspond à une paire d'actions : un *read* effectué par la mémoire principale et un *load* effectué par le thread. L'action *read* lit la valeur dans la mémoire principale et la transmet au thread. L'action *load* accepte la valeur en provenance de la mémoire principale et écrit cette valeur dans le cache du thread. Symétriquement, le transfert dans le sens inverse, depuis le cache du thread vers la mémoire principale correspond a une paire d'actions : *store* effectuée par le thread et *write* effectuée par la mémoire principale. L'action *store* transmet à la mémoire principale la valeur stockée dans

le cache du thread. L'action *write* accepte la valeur transmise par le thread et écrit cette valeur dans la mémoire principale.

Le modèle de mémoire de Java définit également des actions de synchronisation. Les sections critiques de Java (appelées *moniteurs*) reposent sur des verrous. À chaque objet Java est associé un verrou, qui est stocké en mémoire principale. Un verrou peut être possédé par un seul thread à la fois. (Un thread qui demande un verrou bloqué par un autre thread reste en attente jusqu'à ce que ce dernier libère le verrou.) Les threads peuvent exécuter des actions *lock* et *unlock* et la mémoire principale est responsable de l'exécution de ces actions. Lorsqu'un thread obtient un verrou, il doit invalider toutes les variables de son cache local. Avant de relâcher le verrou, il doit d'abord transmettre à la mémoire principale toutes les modifications effectuées sur les variables du cache local depuis le blocage du verrou. Ces deux règles garantissent un fonctionnement correct dans le cas d'une utilisation conventionnelle des verrous pour protéger une variable partagée. Après que le thread bloque le verrou, la première référence à la variable déclenche la lecture de sa valeur dans la mémoire principale est son stockage dans le cache local. La valeur de la variable peut ensuite être modifiée localement et, lorsque le thread libère le verrou, la valeur modifiée est transmise à la mémoire principale, afin qu'elle devienne visible aux autres threads.

5.3.2 Cohérence Java dans Hyperion/DSM-PM²

Détails d'implémentation

Principe général. Le concept de *mémoire principale* de Java est implémenté dans Hyperion à l'aide de DSM-PM², en utilisant une stratégie *à référence* (*home-based*). Chaque page avec tous les objets qu'elle contient est associée à un nœud chargé de maintenir une copie de référence. Initialement, chaque objet est stocké sur son nœud de référence et peut être répliqué s'il est accédé sur d'autres nœuds. Lorsqu'un objet est utilisé dans le membre droit d'une affectation, la copie locale de l'objet est accédée. S'il n'y a pas de copie dans le cache local, une telle copie est chargée depuis la mémoire principale (donc depuis le nœud de référence) par la primitive `loadIntoCache`. L'*affectation* d'un objet détermine la modification de la copie de cet objet dans le cache. Dans cette situation également, une copie de l'objet est préalablement chargée depuis le nœud de référence s'il n'y a pas de copie dans le cache.

Dans le modèle de mémoire de Java, à chaque objet est associé un verrou qui est utilisé pour protéger les accès concurrents des threads par une section critique. L'entrée d'un thread en section critique détermine l'invalidation du cache local, afin d'empêcher le thread d'accéder à des valeurs éventuellement obsolètes. Par conséquent, tout accès à une donnée partagée dont le nœud de référence est distant déclenche le transfert d'une copie de la page correspondante depuis ce nœud vers le nœud local. Lors de la sortie de la section critique, les modifications effectuées localement sont transmises au nœud de référence. L'implémentation est très similaire à celle du protocole `hbrc_mw`, qui implémente la cohérence à la libération : elle utilise une stratégie à référence et permet la coexistence des écrivains multiples. En effet, dans les deux protocoles, des objets différents situés sur la même page peuvent être modifiés de manière concurrente sur différents nœuds, qui transmettent leurs modifications au nœud de référence lors de la sortie de la section critique. Les deux modèles de cohérence sont d'ailleurs proches. Gontchmakher et Schuster [37] ont montré que la cohérence Java est équivalente à la cohérence à la libération pour tous les accès synchronisés à des variables non-volatiles. Une cohérence plus stricte est exigée dans les autres cas.

Stockage et envoi des différences. Hyperion utilise des primitives spécifiques pour accéder aux variables partagées : `get` et `put`. Ceci permet de détecter et d'enregistrer toutes les modifications aux objets du cache local, à l'aide d'un *bitmap*, dans lequel chaque bit correspond à un octet d'un objet. Un bit à 1 signifie que l'octet a été modifié localement. Ce bitmap et les primitives de manipulation associées font partie des structures de données génériques fournies par la table des pages de DSM-PM^2. Hyperion en fait ici une utilisation spécifique : la primitive `put` utilise ce bitmap pour enregistrer les modifications effectuées localement aux objets, avec une granularité variable, égale à la taille des champs des objets. Toutes les modifications locales sont transmises au nœud de référence par la primitive `updateMainMemory`. Lors de cette mise à jour, Hyperion identifie toutes les variables modifiées présentes dans le cache, trouve les nœuds de référence pour toutes ces variables et génère ensuite des RPC pour transmettre les valeurs modifiées. Le thread appelant reste bloqué jusqu'à ce que le nœud de référence réponde par un RPC d'acquittement qui confirme que les modifications ont été reçues et stockées sur le nœud de référence. L'attente de ces acquittements est nécessaire afin d'empêcher le thread appelant de libérer le verrou prématurément. Une telle libération prématurée conduirait à un comportement incorrect sémantiquement si un autre thread bloquait le verrou et chargeait les valeurs des variables présentes sur le nœud de référence, avant que les modifications effectuées par le premier thread soient prise en compte sur le nœud de référence.

Caches au niveau des nœuds. Dans la description originale du modèle de mémoire de Java, tout thread possède son propre cache. Lors de la conception d'Hyperion, on a pris en compte le fait que la performance de beaucoup d'applications peut être améliorée par l'utilisation d'un grand nombre de threads (supérieur au nombre de nœuds) qui s'exécutent concurremment. Afin de supporter efficacement cette approche, Hyperion permet aux threads s'exécutant sur le même nœud de partager le même cache. Ce partage permet d'améliorer l'efficacité des accès grâce à l'effet de *prefetching* dont nous venons de discuter, qui optimise la localité des accès. De plus, la taille occupée par les caches est optimisée également, car un seul cache par nœud est nécessaire, et non pas un cache par thread.

L'exécution concurrente de plusieurs threads par nœud peut mener à l'exécution concurrente d'exécution d'actions de cohérence telles que le chargement des pages, l'invalidation des caches ou la mise à jour de la mémoire principale. Pour gérer correctement ces situations de concurrence, Hyperion utilise des verrous, ainsi que des mécanismes de synchronisation plus complexes.

- Tout d'abord, chaque page du cache local est protégée par un verrou d'exclusion mutuelle qui permet de sérialiser les opérations de mise à jour du bitmap lors des appels des primitives `put`. Ce verrou est également bloqué avant la transmission des modifications stockées dans le bitmap vers le nœud de référence. Le verrou n'est pas bloqué par la primitive `get`, car la lecture d'un champ peut s'exécuter de manière concurrente à la mise à jour du bitmap de la page correspondante ainsi qu'à la transmission des modifications vers le nœud de référence.
- Ensuite, un verrou est utilisé au niveau de chaque nœud afin d'empêcher les invalidations concurrentes du cache ainsi que les mises à jour concurrentes de la mémoire principale. Ce verrou est nécessaire pour protéger les structures de données internes utilisées par le cache.
- Enfin, un mécanisme de synchronisation plus complexe est utilisé pour empêcher tout thread d'effectuer une invalidation du cache jusqu'à ce que tous les threads arrivent à un point où cette invalidation peut s'exécuter. Ce mécanisme est similaire aux solutions au

problème des "lecteurs et des écrivains", car plusieurs threads (lecteurs) peuvent accéder le cache alors qu'un seul thread (écrivain) peut l'invalider à la fois. De plus, aucun thread ne peut accéder le cache pendant qu'il est invalidé. Les threads libèrent le verrou des "lecteurs" dès qu'ils arrivent à un point où l'objet pourrait être déplacé. Le mécanisme empêche l'invalidation du cache pendant le chargement d'une page, afin d'éviter le déchargement éventuel de la page et la ré-émission d'une requête. De plus, des requêtes de page concurrentes peuvent être satisfaites par un seul chargement de la page depuis le nœud de référence.

Synchronisation et cohérence. Lorsqu'un thread exécute une action *lock*, Hyperion exécute d'abord le code pour acquérir le verrou, ensuite il transmet à la mémoire principale toutes les valeurs modifiées présentes dans le cache. Cette opération est réalisée par la primitive `updateMainMemory`. Ensuite, le cache local est invalidé. Lorsque le thread exécute une action *unlock*, Hyperion met d'abord à jour la mémoire principale (toujours via la primitive `updateMainMemory`) et ensuite exécute le code qui correspond à la libération du verrou.

Nous remarquerons que le modèle de mémoire de Java requiert seulement que le cache soit invalidé lors du blocage d'un verrou, alors qu'Hyperion met à jour la mémoire principale également. Cette mise à jour n'est pas requise explicitement par les règles du modèle, mais elle est impliquée par une règle qui exige que toute action *assign* exécutée pour une variable soit suivie d'une action *store* avant toute action *load* concernant cette variable. Si les modifications effectuées localement par le thread n'étaient pas transmises à la mémoire principale (représentée par le nœud de référence) avant l'invalidation du cache, ces valeurs seraient perdues et la règle que nous venons de mentionner ne serait pas respectée.

Implémentation à base de pages. Dans Hyperion, les objets Java sont alloués sur des pages de mémoire virtuellement partagée gérées par DSM-PM^2. Si un objet occupe plusieurs pages, toutes ces pages sont copiées dans le cache local lors du chargement de l'objet depuis le nœud de référence et, inversement, d'autres objets éventuellement situés sur la même page sont chargés. Par conséquent, le chargement d'un objet dans le cache local peut produire un effet de *prefetching*. Ceci est compatible avec la spécification du modèle de mémoire de Java, car les actions prématurées de *read* et *load* sont explicitement permises si elles sont effectuées après la dernière opération *lock* exécutée par le thread. Étant donné que chaque opération *lock* comporte une invalidation du cache local, toute valeur chargée prématurément grâce à cet effet de *prefetching* avant la dernière opération *lock* est invalidée au moment du *lock*. Ceci garantit que toute valeur accédée dans le cache a été forcément chargée après la dernière opération *lock*.

Notons que le fait d'utiliser un cache à base de pages et non pas un cache d'objets n'affecte pas la correction de l'opération de mise à jour de la mémoire principale. Puisque dans le modèle de mémoire de Java les opérations *lock* et *unlock* déclenchent l'envoi des modifications locales de *tous* les objets vers la mémoire principale, l'utilisation d'un cache de pages reste compatible avec le modèle.

Mécanismes alternatifs de détection d'accès à distance

Les principes d'implémentation présentés ci-dessus ont été utilisés à travers deux protocoles de DSM-PM^2 pour la cohérence Java : `java_ic` et `java_pf`. Ces protocoles diffèrent par les mécanismes qu'ils utilisent pour la détection des accès à des objets distants.

Détection d'accès distants par des tests explicites : le protocole `java_ic`. Étant donné qu'Hyperion utilise des primitives spécifiques d'accès aux données partagées, des tests explicites peuvent être utilisés pour détecter si un objet est présent sur le nœud local (c'est-à-dire si une copie de cet objet se trouve dans le cache). Dans ce cas, il n'est plus nécessaire de faire appel au mécanisme générique basé sur des défauts de page utilisé par DSM-PM^2 pour la détection d'accès. Si le test indique que l'objet est présent dans le cache, cet objet est directement accédé. Sinon, une copie de l'objet est transférée depuis le nœud de référence. Ce schéma est utilisé par le protocole `java_ic` (où `ic` signifie *in-line check*, en français *test en ligne*). Son principal avantage est d'éliminer le coût des défauts de page qui seraient nécessaires pour détecter les accès non-locaux si on utilisait le mécanisme par défaut de DSM-PM^2. Par conséquent, aucune protection de page n'est nécessaire, puisque tous les accès aux données partagées sont effectuées à travers les primitives `get` et `put`. La mémoire partagée est accessible et lecture et en écriture sur tous les nœuds dès l'initialisation et ces droits d'accès restent ainsi jusqu'à la fin de l'application. En conclusion, les accès à des objets distants sont moins coûteux, car ils ne déclenchent pas de défaut de page, ni d'appel à la primitive `mprotect` pour la gestion les droits d'accès. En contrepartie, un test explicite de localité est nécessaire lors de *chaque* accès à un objet, qu'il soit local ou distant.

Détection d'accès distants par défauts de page : le protocole `java_pf`. Une autre méthode de détection d'accès distants consiste à utiliser le mécanisme des défauts de page de DSM-PM^2. Cette solution est implémentée par le protocole `java_pf` (où `pf` signifie *page faults*). Initialement, les objets sont accessibles en lecture et écriture sur le nœud de référence et restent protégés de tout type d'accès sur les autres nœuds. Cette protection est activée lors de chaque entrée en section critique, de manière à ce que tout accès à un objet sur un nœud différent du nœud de référence produise un défaut de page. Ce défaut de page déclenche le protocole de cohérence, qui demande le transfert d'une copie de la page contenant l'objet depuis le nœud de référence. Lors de l'arrivée de la page, les droits d'accès correspondants sont positionnés pour permettre l'accès en lecture et en écriture et le thread ayant généré le défaut de page est ainsi assuré d'accéder des valeurs à jour, une fois entré en section critique. Le coût des accès distant est donc plus élevé dans le cas de ce protocole, à cause du surcoût introduit par le défaut de page et par les appels à la primitive Unix `mprotect`. En revanche, contrairement au protocole `java_ic`, les accès aux objets locaux (c'est-à-dire aux objets présents dans le cache local ou accédés sur leur propre nœud de référence) sont moins coûteux, car aucun test de localité n'est plus nécessaire.

A priori, le choix entre une stratégie ou l'autre implique un compromis qui doit prendre en compte des facteurs tels que le rapport entre le nombre d'accès locaux et le nombre d'accès distants (qui est déterminé par la distribution des données et par le rapport communication/calcul), ainsi que le coût relatif des défauts de page par rapport au coût des tests. Grâce à la flexibilité de DSM-PM^2, nous avons pu expérimenter les deux techniques en les intégrant à deux protocoles de cohérence.

5.4 Évaluation des performances : quelques applications

5.4.1 Applications utilisées

Nous avons évalué nos deux protocoles à l'aide de 5 applications à caractéristiques différentes.

- L'application `Pi` calcule une estimation de π comme une somme Riemann de 50 millions de valeurs. Ce programme présente un haut degré de parallélisme, car les threads se synchronisent seulement pour calculer la somme globale à partir des sommes partielles calculées localement par les threads pour leurs intervalles Riemann respectifs.
- L'application `Jacobi` calcule la distribution de la température sur une plaque isolée, après 100 pas de calcul, en utilisant une grille de cellules 1024×1024 pour discrétiser la plaque. Chaque thread possède un bloc de lignes contiguës de la grille. À chaque pas de calcul , chaque thread doit récupérer deux lignes de "frontière" : une de son voisin qui possède les lignes situées au "nord" des lignes locales et une autre du voisin du "sud".
- `Barnes Hut` est une application de simulation d'interaction gravitationnelle de N particules (*N-body*), adaptée en Java à partir d'un code C distribué fourni par les benchmarks SPLASH-2. Pour nos expérimentations, nous avons utilisé 16K particules et 6 pas de calcul. Le schéma des communications est irrégulier, car les particules changent de position pendant la simulation, ce qui modifie également les interactions des particules d'un pas de calcul à l'autre. L'application utilise un algorithme d'équilibrage de charge qui distribue dynamiquement les particules aux threads, à chaque pas de calcul.
- `TSP` implémente une solution de type *branch-and-bound* au problème du voyageur de commerce (*Travelling Salesperson Problem*) qui calcule le chemin le plus court passant exactement une fois par toutes les villes d'un ensemble donné. Nous avons résolu un problème de taille 17. Le programme utilise une file centrale contenant les tâches à exécuter, ainsi qu'une variable qui stocke la meilleure solution courante. Ces structures de données "centrales" sont stockées sur un nœud, sont protégées par un moniteur Java et sont accédées à distance par les threads qui s'exécutent sur les autres nœuds.
- `ASP` (All-pairs, Shortest Paths) calcule le plus court chemin entre toutes les paires de nœuds d'un graphe, en utilisant l'algorithme de Floyd. Nous avons utilisé un graphe à 2000 nœuds. `ASP` utilise une matrice de distances à 2 dimensions. Comme dans `Jacobi`, chaque thread possède un bloc de lignes contiguës. À chaque itération, la ligne "courante" de la matrice est lue par tous les threads.

(Les applications `TSP` et `ASP` nous ont été gracieusement fournies par le groupe Jackal de Vrije Universiteit.)

5.4.2 Résultats expérimentaux

Les applications ont été exécutées sur deux plates-formes. La première a été constituée de 12 PC Pentium Pro à 200 MHz sous Linux 2.2.13 reliés par un réseau Myrinet sous BIP. (Étant donné que la grappe PC PII à 450 MHZ sous BIP à laquelle nous avons eu accès ne comportait que 4 nœuds disponibles, nous avons choisi d'utiliser cette plate-forme à processeurs moins performants, mais qui propose 12 nœuds). La deuxième est formée de 6 PC Pentium II à 450 MHz sous Linux 2.2.13 interconnectés par un réseau SCI exploitée par l'interface SISCI.

Le compilateur `gcc` (version 2.7.2.3) a été intégré dans la chaîne de compilation d'Hyperion, pour compiler le code généré par `java2c`. Le compilateur `gcc` a été invoqué avec l'option de compilation `-O6`.

Les figures 5.4 à 5.8 présentent les performances relatives des deux implémentations pour chaque application testée, sur chacune des deux grappes. Chaque application crée un thread de calcul par processeur de la grappe. Les accélérations correspondantes sont données dans les tables 5.2 et 5.3.

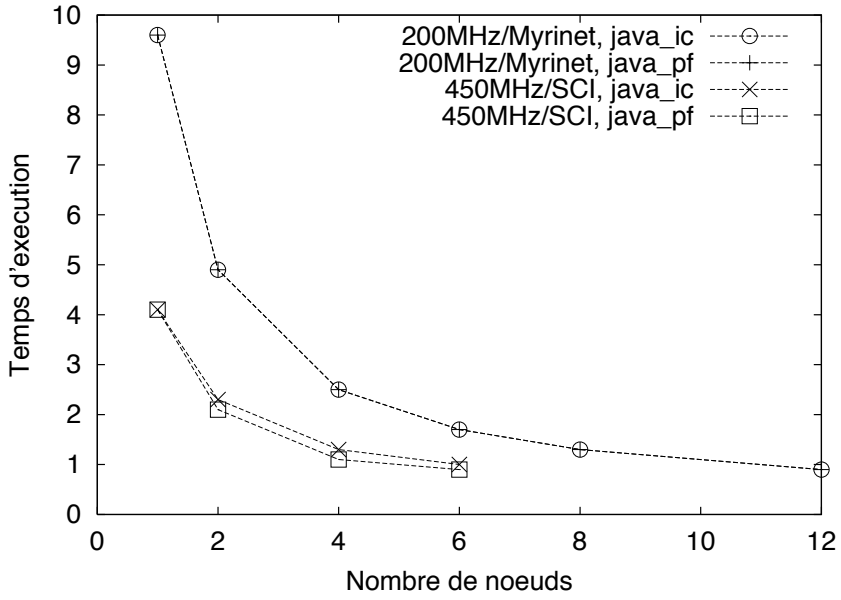


FIG. 5.4 – Pi : `java_pf` *vs* `java_ic`.

5.4.3 Discussion

Dans le cas de l'application `Pi`, les deux protocoles ont un comportement pratiquement identique sur les deux grappes. Ceci n'est pas surprenant, car les threads du programme utilisent très peu les objets, le calcul s'effectuant presqu'exclusivement avec des valeurs stockées dans la pile des threads. Dans cette situation, `java_ic` exécute très peu de tests de localité et `java_pf` déclenche très peu de défaut de pages, ce qui conduit globalement à des performances très proches.

Dans le cas des autres applications, sur la grappe Myrinet, le protocole `java_pf` permet d'obtenir des performances nettement supérieures à celles produites par `java_ic`. L'amélioration obtenue se situe entre 38% pour `Jacobi` et 64% for `ASP`. Le plus important facteur qui détermine cette amélioration est le coût relatif du test explicite utilisé par le protocole `java_ic` par rapport au coût du reste du calcul effectué par l'application. Dans `ASP`, la boucle la plus interne effectue simplement une addition d'entiers et une comparaison d'entiers, tout en effectuant trois tests de localité. La suppression de ces tests dans le protocole `java_pf` apporte clairement une amélioration significative des performances. À titre de comparaison, la boucle la plus interne de `Jacobi` utilise des opérations double précision sur des flottants, plus coûteuses, ce qui rend le coût des tests de localité moins important.

Dans le cas des applications `Jacobi`, `TSP` et `ASP`, l'amélioration reste relativement constante sur la grappe Myrinet, lorsqu'on fait varier le nombre de nœuds. En revanche, pour `Barnes`, l'amélioration passe de 46% à 28% lorsque le nombre de nœuds passe de 1 à 12. Pour cette application et pour la taille de problème fixée, le temps d'exécution reste relativement constant lorsque le nombre de nœuds est élevé, car le coût des communications augmente et devient prédominant. Par conséquent, le nombre de défauts de page gérés par `java_pf` ainsi que le nombre d'appels à la primitive `mprotect` augmentent de manière significative. Ce surcoût compense l'amélioration obtenue par l'élimination des tests de localité. Nous remarquerons par ailleurs que l'accélération

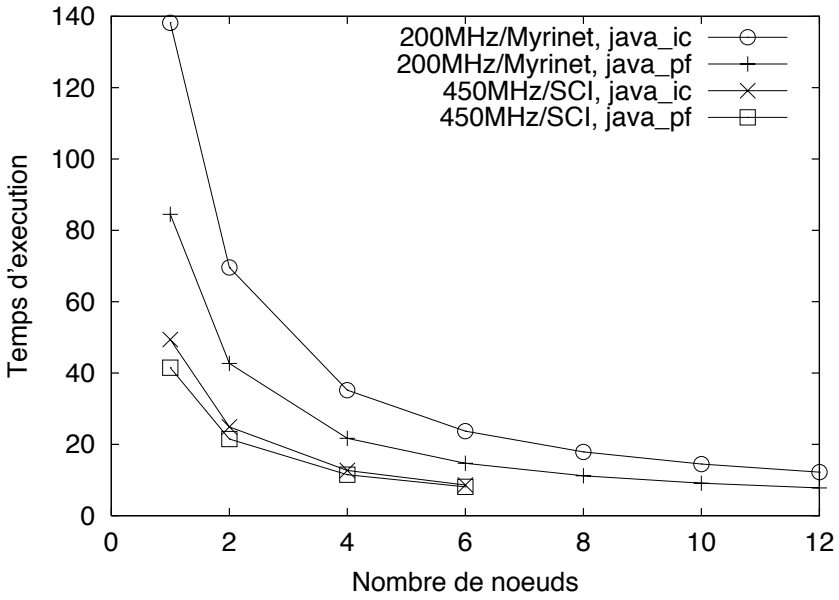


FIG. 5.5 – Jacobi : `java_pf` *vs* `java_ic`.

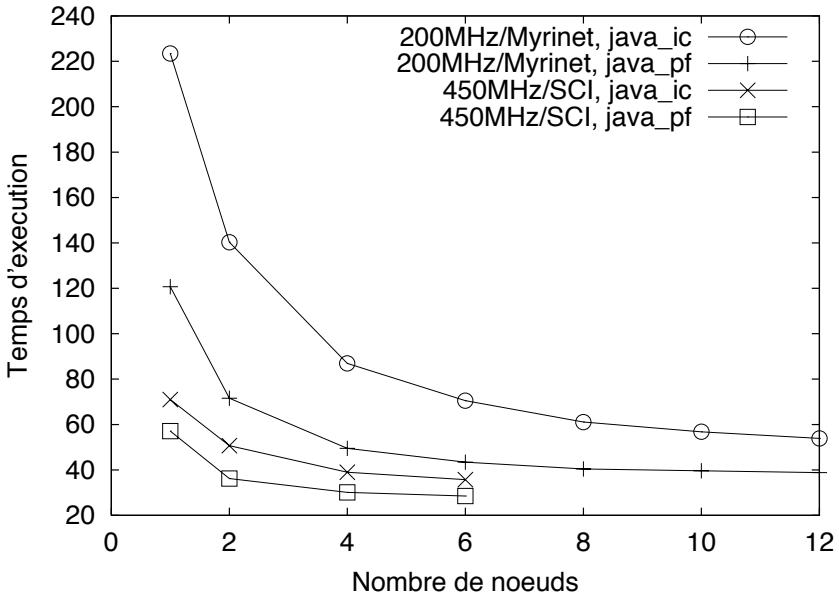


FIG. 5.6 – Barnes Hut : `java_pf` *vs* `java_ic`.

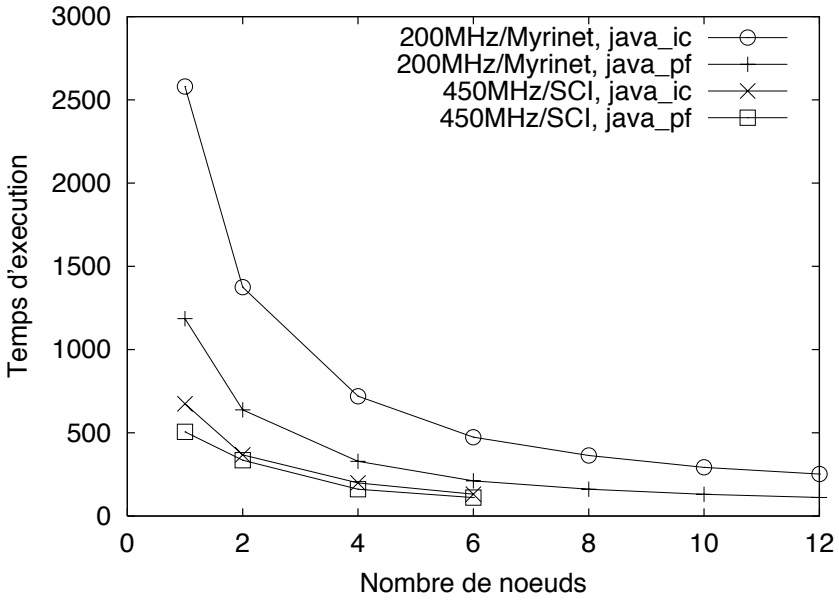


FIG. 5.7 – TSP : `java_pf` *vs* `java_ic`.

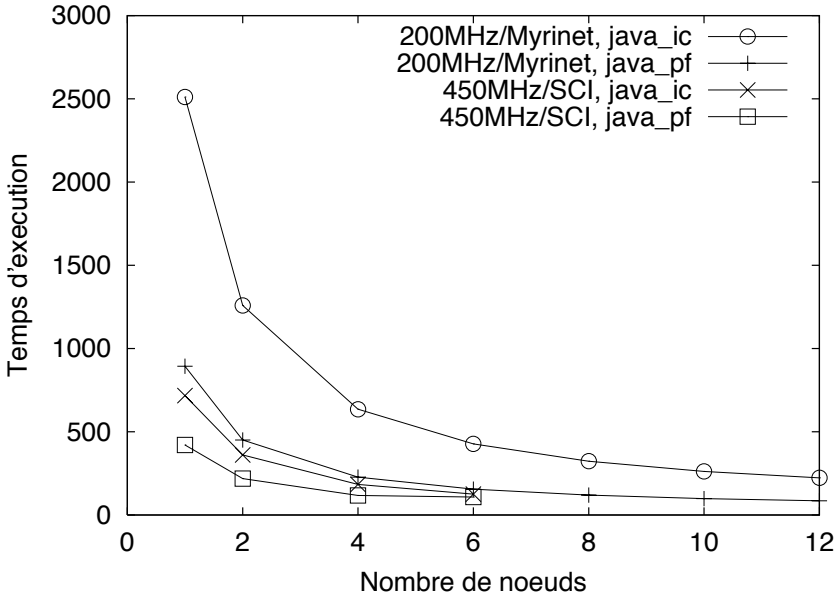


FIG. 5.8 – ASP : `java_pf` *vs* `java_ic`.

	Nb. nœuds :	2	4	6	8	10	12
Pi	`java_ic`	1.96	3.84	5.65	7.38	9.6	10.66
	`java_pf`	1.96	3.84	5.65	7.38	9.6	10.66
Jacobi	`java_ic`	1.99	3.93	5.83	7.72	9.53	11.32
	`java_pf`	1.98	3.89	5.75	7.54	9.29	10.83
Barnes Hut	`java_ic`	1.59	2.57	3.17	3.66	3.93	4.15
	`java_pf`	1.69	2.44	2.78	2.99	3.05	3.11
TSP	`java_ic`	1.88	3.58	5.45	7.09	8.83	10.23
	`java_pf`	1.86	3.60	5.60	7.35	9.07	10.61
ASP	`java_ic`	1.99	3.95	5.88	7.78	9.60	11.26
	`java_pf`	1.98	3.93	5.74	7.46	9.07	10.49

TAB. 5.2 – Accélérations obtenues sur une grappe Myrinet à 12 nœuds PPro à 200 MHz.

obtenue pour cette application n'est satisfaisante sur aucune des deux grappes (cf. tables 5.2 et 5.3), ce qui n'est pas le cas des autres applications.

Le coût des communications pour `Jacobi`, `TSP` et `ASP` est relativement constant sur la grappe Myrinet pour les tailles des problèmes étudiées, lorsqu'on fait varier le nombre de nœuds. Pour `Jacobi`, ce coût est constant ; pour `TSP` et `ASP`, le coût des communications est caché grâce aux recouvrement par des calculs, même pour un grand nombre de nœuds. Le surcoût dû aux défauts de page et aux opérations de protection de la mémoire n'est pas significatif dans cette situation.

Sur la grappe SCI aussi, le protocole `java_pf` produit une meilleure performance que `java_ic` pour les applications `Jacobi`, `Barnes`, `TSP` et `ASP`. Néanmoins, l'amélioration est moins importante que sur la grappe Myrinet : nous avons pu mesurer un gain moyen de 21% sur l'ensemble des applications. Ceci est dû essentiellement à la fréquence supérieure de l'horloge dans le cas des PC de la grappe SCI, qui rend l'élimination des tests explicites moins importante.

L'ensemble de ces résultats montre que le protocole basé sur des défauts de page est supérieur pour toutes les applications sur les deux grappes étudiées. Pour des applications parallèles qui présentent un bonne accélération lorsqu'on augmente le nombre de nœuds, les surcoût dus aux défauts de page ne sont pas significatifs, car le coût des communications n'est pas dominant pour ce type d'applications. Dans ce cas, l'amélioration observée sur un seul nœud grâce à l'élimination des tests de localité devrait pouvoir être observée également pour des exécutions sur un grand nombre de nœuds (par exemple, supérieur à 8). L'amélioration obtenue dépendra du coût relatif des tests par rapport au coût du reste des calculs. D'autre part, notons que nous avons utilisé un seul thread par nœud pour nos expérimentations. Nous nous sommes également proposé d'étudier les améliorations supplémentaires qui pourraient être obtenues en exécutant nos applications avec plusieurs threads par nœuds, car ceci favoriserait un meilleur recouvrement des communications par des calculs. Les tests préliminaires que nous avons effectués nous encouragent à poursuivre cette direction.

5.5 Conclusion

La plate-forme DSM-PM^2 a permis au projet Hyperion de proposer une infrastructure logicielle d'exécution *transparente* des programmes multithreads Java sur des grappes de PC à hautes

	Nb. nœuds :	2	4	6
Pi	`java_ic`	1.78	3.15	4.1
	`java_pf`	1.95	3.72	4.56
Jacobi	`java_ic`	1.98	3.89	5.74
	`java_pf`	1.93	3.61	5.12
Barnes Hut	`java_ic`	1.40	1.82	1.99
	`java_pf`	1.58	1.90	2.00
TSP	`java_ic`	1.83	3.39	5.13
	`java_pf`	1.51	3.14	4.56
ASP	`java_ic`	1.99	3.91	5.73
	`java_pf`	1.92	3.59	3.89

TAB. 5.3 – Accélérations obtenues sur une grappe SCI à 6 nœuds PII à 450 MHz.

performances et à faible coût. Hyperion présente au programmeur la vue d'une machine virtuelle Java unique et cache les détails liés à la distribution des données et des traitements. Par conséquent, des applications multithreads Java existantes écrites pour un environnement à mémoire *physiquement* partagée peuvent s'exécuter *sans modification* sur plusieurs nœuds. Hyperion met en œuvre l'exécution distribuée en projetant les threads Java sur des threads natifs de la grappe, qu'il distribue sur les différents nœuds en vue d'une exécution parallèle, plus efficace. Le modèle de mémoire de Java est implémenté au-dessus de l'interface proposée par le système de mémoire virtuellement partagée DSM-PM^2, tout en respectant la sémantique originale du langage. Ceci représente une application typique de la plate-forme DSM-PM^2, qui fournit les mécanismes de base pour l'implémentation de protocoles de cohérence multithreads.

La portabilité a été un objectif majeur lors de la conception d'Hyperion. L'implémentation actuelle ne dépend d'aucune spécificité propre à un système d'exploitation, ni d'une architecture particulière, ni d'un réseau, ni d'une interface de communication particulière. Cet objectif a pu être atteint grâce à l'implémentation au-dessus de PM^2 et de DSM-PM^2, qui ont fourni "gratuitement" au système complet la portabilité sur un grand nombre de types de grappes, basées sur des réseaux tels que SCI, Myrinet, ou Fast-Ethernet et d'être utilisé avec les interfaces de communication et les protocoles les plus répandus. Hyperion a été testé sur des grappes de PC sous Linux, mais le système a été conçu pour être fonctionnel sur tout système d'exploitation de type Unix.

Nos résultats expérimentaux mettent en évidence cette portabilité, car les mêmes applications ont pu être exécutées *sans aucune modification* sur des grappes à processeurs différents, utilisant des technologies réseau différentes et des protocoles de communication différents. La flexibilité de la plate-forme DSM-PM^2 a permis la mise en œuvre de deux protocoles de cohérence basés sur des mécanismes différents de détection d'accès distants. Ceci a permis d'identifier un facteur important qui améliore la performance de l'exécution des codes Java : l'élimination des tests de localité. Enfin, nous avons présenté des mesures de performance qui démontrent une bonne efficacité lors de l'exécution parallèle des applications multithreads Java compilées avec Hyperion et exécutées sur plusieurs grappes grâce à DSM-PM^2. Cette implémentation constitue une validation de DSM-PM^2 en tant que plate-forme d'implémentation de protocoles de cohérence multithreads, intégrée dans un système de compilation.

Chapitre 6

Conclusion

Ce chapitre résume les propositions présentées dans cette thèse et examine les principales perspectives ouvertes par ce travail.

6.1 Travaux réalisés

La contribution de cette thèse se traduit par la proposition d'une notion de *mémoire virtuellement partagée générique* dans un contexte *multithread* et par la réalisation de la plate-forme DSM-PM^2, qui illustre cette notion. Nos travaux ont permis de mettre en évidence des problématiques associées à la conception de protocoles de cohérence multithreads, que nous avons abordées lors de la conception de 6 protocoles de cohérence multithreads intégrés dans la bibliothèque de la plate-forme.

Mémoire virtuellement partagée générique. Nous avons défini la notion de *mémoire virtuellement partagée générique* suite à l'étude de plusieurs environnements à mémoire virtuellement partagée existants. Nous nous sommes limité à des systèmes à base de pages, implémentés entièrement au niveau logiciel, en espace utilisateur. À l'issue de cette analyse, nous avons identifié un certain nombre de mécanismes *génériques* de base, présents dans la plupart des systèmes étudiés, indépendants des modèles de cohérence mis en œuvre et potentiellement réutilisables lors de la conception de nouveaux protocoles. Il s'agit de mécanismes de gestion des communications ou de gestion la table des pages partagées, ou encore du mécanisme d'association d'actions de cohérence *spécifiques* à un protocole particulier à des événements *génériques*, tels qu'un défaut de page, la réception d'une requête de page, etc.

Plate-forme d'implémentation de protocoles de cohérence multithreads. Nous avons illustré la notion de MVP générique par la réalisation de la plate-forme DSM-PM^2. Cette plate-forme a plusieurs particularités.

Généricité. DSM-PM^2 est avant tout une *plate-forme d'implémentation et d'expérimentation* de protocoles de cohérence, construits à l'aide des éléments générique. C'est une plate-forme *configurable* et *extensible* : le programmeur a la possibilité d'associer explicitement un protocole disponible dans la bibliothèque de la plate-forme à une donnée et de définir de nouveaux protocoles, qui peuvent être utilisés de la même manière que les protocoles intégrés.

Multithreading. DSM-PM^2 a été réalisé dans le cadre de l'environnement *multithread* PM^2. D'une part, l'implémentation de la plate-forme utilise les threads de manière interne ; d'autre part, la plate-forme supporte le multithreading au niveau utilisateur : toute application peut générer un nombre arbitraire de threads et ces threads peuvent lire et écrire dans la mémoire commune quelle que soit leur localisation physique sur la grappe.

Portabilité. La plate-forme est *portable* sur plusieurs systèmes d'exploitation de type Unix (Linux, Solaris, AIX) et sur un grand nombre de réseaux. Cette dernière propriété est due à l'implémentation de l'ensemble des routines génériques de communication par des RPC PM^2, qui à leur tour reposent sur la bibliothèque de communication MADELEINE. Par conséquent, DSM-PM^2 supporte l'ensemble des réseaux supportés par MADELEINE (SCI, Myrinet, Fast Ethernet) et l'ensemble des interfaces de communications associées : SISCI, BIP, TCP, MPI, VIA.

Allocation et migration iso-adresse. Nous avons introduit le concept d'*allocation et migration iso-adresse* dans PM^2 à travers la conception d'un allocateur dynamique permettant de garantir le stockage des données à la même adresse virtuelle. Initialement créé pour les piles des threads et leurs données privées, ce mécanisme a été étendu pour DSM-PM^2 afin de permettre l'allocation dynamique et la migration iso-adresse des *pages partagées*. Par conséquent, les threads peuvent manipuler en toute sécurité des pointeurs vers des données partagées : un pointeur référencera toujours la même donnée sur tous les nœuds.

Protocoles de cohérence multithreads. Afin de valider la généricité de DSM-PM^2, nous avons implémenté, dans un premier temps, deux modèles de cohérence à caractéristiques différentes : un modèle de cohérence forte (la cohérence séquentielle) et un modèle de cohérence faible (la cohérence à la libération). Pour chaque modèle, nous avons réalisé deux protocoles basés sur des mécanismes sous-jacents différents. La cohérence séquentielle est implémentée par un protocole à base de migration/réplication de pages dérivé d'un protocole monothread classique et par un deuxième protocole original basé sur la migration de threads. Deux autres protocoles (réalisés en collaboration avec Vincent Bernardi) implémentent la cohérence à la libération : un protocole à écrivain unique et un autre à écrivains multiples. La conception et l'implémentation de l'ensemble de ces protocole a nécessité la prise en compte de plusieurs situations de concurrence spécifiques au contexte multithread. Ces situations de concurrence augmentent de manière significative la complexité des protocoles de cohérence. Dans les solutions que nous avons proposées, nous nous sommes fixé un double objectif : d'une part, garantir la correction du protocole, d'autre part, assurer un degré de concurrence suffisamment élevé, qui permette une exécution efficace. Les protocoles ont été validés expérimentalement à l'aide de deux applications typiques.

Intégration dans un système de compilation Java pour grappes. L'un de nos premiers objectifs a été de proposer une plate-forme utilisable au sein des systèmes de compilation de langages basés sur le paradigme de la mémoire partagée. Nous avons validé cet objectif en spécialisant DSM-PM^2 pour une utilisation dans le système Hyperion de compilation Java pour grappes. Ce travail a été effectué en collaboration avec Philip Hatcher (University of New Hampshire, NH, USA). La flexibilité de la plate-forme et ses mécanismes génériques ont permis l'implémentation de deux protocoles de cohérence différents, spécifiques pour le modèle de mémoire de Java. Grâce à cette approche, des programmes multithreads Java

peuvent s'exécuter *inchangés* sur des machines monoprocesseurs ou sur des grappes basées sur différents réseaux et interfaces de communication. Les expérimentations effectuées avec 5 applications multithreads Java ont permis de mesurer de très bonnes performances et un bon passage à l'échelle lorsqu'on varie le nombre de nœuds.

6.2 Perspectives

Conçu pour une utilisation comme plate-forme expérimentale, DSM-PM^2 fournit à ce jour un grand nombre de mécanismes de base dont nous avons présenté différentes utilisations à travers les protocoles de cohérence mis en œuvre jusqu'à présent. D'autres développements peuvent être envisagés dans l'avenir, dans plusieurs directions.

Compilation OpenMP. Le langage OpenMP est en train de s'imposer aujourd'hui comme standard pour la programmation parallèle basée sur le paradigme de la mémoire partagée. Dans ce contexte, la spécialisation de la plate-forme DSM-PM^2 en vue d'une intégration dans un système de compilation OpenMP pour grappes devient une application intéressante, qui faciliterait la réalisation d'expérimentations de différentes variantes d'implémentation, sur différents types de grappes. Ce travail (similaire à celui que nous avons effectué dans le cadre du projet Hyperion) permettrait également de renforcer la généricité de la plate-forme. Sur ce thème, une collaboration avec Christian Pérez (projet PARIS, IRISA, Rennes) est envisagée.

Protocoles adaptatifs. Nous avons vu que la DSM-PM^2 permet d'implémenter et d'expérimenter des protocoles différents pour le même modèle de cohérence. Le choix d'un protocole pour chaque donnée partagée est aujourd'hui à la charge du programmeur. Dans la plupart des cas, un protocole n'est jamais supérieur *dans toutes les situations* à un autre protocole qui implémente le même modèle de cohérence par une stratégie différente. Il est alors intéressant d'analyser finement les performances de chaque protocole dans toutes les situations et de mettre en œuvre des protocoles *adaptatifs* qui choisissent dynamiquement la stratégie de cohérence la plus performante. Le choix de la "bonne" stratégie peut dépendre de plusieurs critères tels que la taille de la donnée ou le schéma d'accès des threads aux données.

Support pour d'autres modèles de cohérence. Dans sa version courante, la plate-forme supporte trois modèles de cohérence : la cohérence séquentielle, la cohérence à la libération et la cohérence Java. Nous pensons que les mécanismes génériques disponibles permettent l'implémentation d'autres modèles à caractéristiques proches. Un exemple est la *cohérence de portée*, à laquelle de nombreuses études récentes ont été consacrées et qui permet parfois d'obtenir des performances supérieures à celles fournies par la cohérence à la libération.

Cohérence hiérarchiques et protocoles multi-niveaux. Par ailleurs, nous nous sommes focalisé jusqu'à présent sur des modèles de cohérence basés sur une organisation "plate" de la mémoire : tous les nœuds sont équivalents. Une direction de recherche qui nous semble intéressante consiste à prendre en compte des modèles de mémoire hiérarchiques, et de concevoir des protocoles adaptés pour la programmation efficace des architectures émergentes telles que les grappes de grappes de PC [6]. En effet, le coût des communications entre des machines faisant partie de grappes différentes est plus élevé que celui des communications entre machines de la même grappe. Par conséquent, de nouveau protocoles doivent être conçus pour ce type de plate-forme, le principal objectif étant de minimiser les communications inter-grappes et de favoriser le partage intra-grappe. La conception de protocoles

de cohérence efficaces pour ce type d'architecture, ainsi que la prise en compte des autres problématiques associées, telles que la tolérance aux fautes ou l'hétérogénéité, permettra la réalisation d'un partage de données à grande échelle.

Exploitation des nœuds multiprocesseurs. Nos expérimentations avec DSM-PM^2 ont été effectuées sur des machines monoprocesseurs. Néanmoins, lors des choix d'implémentation de la plate-forme, nous avons pris en compte l'évolution de l'environnement PM^2 vers les architectures multiprocesseurs. La bibliothèque de threads de PM^2 (MARCEL) est depuis peu opérationnelle en mode SMP. Ce mode, qui est déjà fonctionnel dans la version courante de MARCEL, repose sur un ordonnancement mixte des threads : un ordonnancement des threads MARCEL en espace utilisateur sur des processeurs virtuels représentés par des threads natifs, ordonnancés par le système d'exploitation sur les processeurs physiques. Ceci permet l'exploitation des nœuds multiprocesseurs d'une manière transparente pour le programmeur. Nous nous sommes proposé d'expérimenter ce mode de fonctionnement, afin d'évaluer sa capacité d'améliorer encore plus les performances des applications exécutées sur DSM-PM^2. L'effort nécessaire est minimal, car il suffit d'utiliser les modèles de cohérence "plats" fournis par la plate-forme, tout en configurant la bibliothèque de threads de PM^2 (MARCEL) pour un fonctionnement en mode SMP.

Spécification et validation formelle. Un autre aspect important concerne la correction des protocoles de cohérence. La validation expérimentale des protocoles implémentés est, certes, nécessaire, mais une validation formelle est souhaitable. Ce type de validation est assez souvent omise dans les travaux qui présentent l'implémentation de systèmes à mémoire virtuellement partagée. Elle peut être envisagée pour DSM-PM^2 si la plate-forme est interfacée avec un environnement de spécification formelle permettant la réalisation de vérifications automatiques de propriétés sur le protocole spécifié. Plusieurs efforts de recherche ont été effectués dans ce domaine, basés sur l'utilisation de différentes techniques de vérification telles que l'énumération d'états ou le model-checking [85], appliquées aux protocoles de cohérence pour MVP(Murϕ [24], SMV [70]). Une approche intéressante proposée dans le thèse de David Mentré [71] (IRISA, Rennes), consiste à utiliser une spécification formelle de haut niveau sur laquelle on puisse vérifier des propriétés et à partir de laquelle on puisse générer un protocole exécutable. Nous pensons que ce type d'approche (certes, complexe !) pourrait être utilisée pour valider la correction des protocoles de cohérence implémentés sur DSM-PM^2.

Bibliographie

[1] S. Ahuja, N. Cariero et D. Gelernter. – Linda and friends. *Computer*, vol. 19, n° 8, mai 1986, pp. 26–34.

[2] AMD. – 3DNow! (TM) technology. – Disponible à l'URL `http://www.amd.com/products/cpg/3dnow/index.html`.

[3] C. Amza, A. L. Cox, S. Dwarkadas, L.J. Jin, K. Rajamani et W. Zwaenepoel. – Adaptive protocols for software distributed shared memory. *Proc. of IEEE*, avril 1999. – Special Issue on Distributed Shared Memory.

[4] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu et W. Zwaenepoel. – TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, vol. 29, n° 2, février 1996, pp. 18–28.

[5] C. Amza, A. L. Cox, S. Dwarkadas et W. Zwaenepoel. – Software DSM protocols that adapt between single writer and multiple writer. *In : Proc. of the Third High Performance Computer Architecture Conference (HPCA'97)*, pp. 261–271. – février 1997.

[6] L. Arantes. – *Conception et réalisation d'un support à mémoire partagée répartie pour grappes de stations inter-connectées*. – Paris, France, Thèse de doctorat, Université Pierre et Marie Curie - Paris 6, 2000.

[7] L. Arantes, P. Sens et B. Folliot. – The impact of caching in a loosely-coupled clustered software dsm system. *In : Proc. IEEE International Conference on Cluster Computing*, pp. 27–34. – Chemnitz, Germany, novembre 2000.

[8] Y. Aridor, M. Factor et A. Teperman. – cJVM: A single system image of a JVM on a cluster. *In : Proc. of the International Conference on Parallel Processing*, pp. 4–11. – Fukushima, Japan, septembre 1999.

[9] H. Bal et A. Tanenbaum. – Distributed programming with shared data. *In : Proc. Int'l Conf. Computer Languages' 88*. pp. 82–91. – Los Alamitos, CA, 1988.

[10] B. N. Bershad, T. E. Anderson, E. D. Lazowsak et H. M. Levy. – Lightweight remote procedure call. *ACM Transactions on Computer Systems*, vol. 8, n° 1, février 1990, pp. 37–55.

[11] B. N. Bershad, M. J. Zekauskas et W. A. Sawdon. – The Midway distributed shared memory system. *In : Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring' 93)*, pp. 528–537. – février 1993.

[12] A. Bilas, L. Iftode et J. P. Singh. – Supporting a coherent shared address space across SMP nodes: An application-driven investigation. *In : IMA Volumes in Mathematics and its Applications*. pp. 19–59. – Springer-Verlag New York Inc., novembre 1998.

[13] R. Bisani et M. Ravishankar. – PLUS: a distributed shared memory system. *In : Proc. 17th Ann. Int'l Symp. Computer Architecture*. pp. 115–124. – Los Alamitos, CA, 1990.

[14] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic et W.-K. Su. – Myrinet : A gigabit-per-second local area network. *IEEE-Micro*, vol. 15, n° 1, février 1995, pp. 29–36.

[15] Luc Bougé, Phil Hatcher, Raymond Namyst et Christian Perez. – A multithreaded runtime environment with thread migration for a HPF data-parallel compiler. *In : The 1998 Intl Conf. on Parallel Architectures and Compilation Techniques (PACT '98)*. IFIP WG 10.3 and IEEE, pp. 418–425. – Paris, France, octobre 1998. Disponible à l'URL `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-43.ps.Z`.

[16] T. Brandes et F. Zimmermann. – Adaptor : A transformation tool for HPF programs. *In : Programming Environments for Massively Parallel Distributed Systems*, éd. par K. M. Decker et R. M. Rehmann. pp. 91–96. – Birkhäuser Verlag, avril 1994.

[17] F. Breg, S. Diwan, J. Villacis, Jayashree Balasubramanian, Esra Akman et Dennis Gannon. – Java RMI performance and object model interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, vol. 10, n° 11-13, 1998, pp. 941–955.

[18] D. Caromel, W. Klauser et J. Vayssière. – Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, vol. 10, n° 11-13, 1998, pp. 1043–1062.

[19] J. B. Carter. – Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, vol. 29, 1995, pp. 219–227. – Special issue on Distributed Shared Mmory.

[20] J. Casas, R. Konuru, S. W. Otto, R. Prouty et J. Walpole. – Adaptive load migration systems for PVM. *In : Proc. Supercomputing '94*, pp. 390–399. – Washington, D. C., novembre 1994.

[21] X. Chen et V. Allan. – MultiJav: A distributed shared memory system based on multiple Java virtual machines. *In : Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 91–98. – Las Vegas, Nevada, juin 1998.

[22] M. de Castro et C. de Amorim. – Efficient categorization of memory sharing patterns in software DSM systems. *In : Proc. 15th Int'l Parallel and Distribued Symp. IPDPS'2001*. p. 63. – San Francisco, CA, avril 2001. Version complète disponible sous forme éléctronique uniquement.

[23] G. Delp, D. Farber et R. Minnich. – Memory as a network abstraction. *IEEE Network*, juillet 1991, pp. 34–41.

[24] D. L. Dill, A. J. Drexler, A. J. Hu et C. Han Jang. – Protocol verification as a hardware design aid. *In : Proc. Int'l Conf. on Computer Design, VLSI in Computers and Processors*. pp. 522–525. – Los Alamitos, CA, octobre 1992.

[25] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee et Y.-S. Chen. – Delayed consistency and its effects on the miss rate of parallel programs. *In : Proc. Supercomputing' 91*, pp. 197–206. – 1991.

[26] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merrit, E. Gronke et C. Dodd. – The Virtual Interface Architecture. *IEEE Micro*, vol. 18, n° 2, mars 1998, pp. 66–76.

[27] S. Dwarkadas, K. Gharachorloo, L. Kontothanassis, D. J. Scales, M. L. Scott et R. Stets. – Comparative evaluation of fineand coarse-grain approaches for software distributed shared memory. *In : Proc. of the 5th Intl. Symp. on High Performance Computer Architecture*, pp. 260–269. – Orlando, FL, janvier 1999.

[28] A. Ferrari. – JPVM: Network parallel computing in Java. *Concurrency: Practice and Experience*, vol. 10, n° 11-13, 1998, pp. 985–992.

[29] B. Fleisch et G. Popek. – Mirage: A coherent distributed shared memory design. *In : Proc. 14th ACM Symp. Operating System Principles*. pp. 211–223. – New York, NY, 1989.

[30] M. J. Flynn. – Some computers organizations ans their effectiveness. *IEEE Transactions on Computer*, vol. 21, n° 9, 1972, pp. 948–960.

[31] I. Foster, C. Kesselman, R. Olson et S. Tuecke. – *Nexus : An interoperability toolkit for parallel and distributed computer systems*. – Rapport technique n° ANL/MCS-TM-189, Argonne National Laboratory, USA, 1994.

[32] S. Frank, H. Burkhardt III et J. Rothnie. – The KSR1: Bridging the gap between shared memory and MPPs. *In : Proc. COMPCON '93*. pp. 285–294. – Los Alamitos, CA, 1993.

[33] R. Friedman, M. Goldin, A. Itzkovitz et A. Schuster. – Millipede : Easy parallel programming in available distributed environments. *Software : Practice and Experience*, vol. 27, n° 8, août 1997, pp. 929–965.

[34] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck et V. Sunderam. – *PVM3 User's guide and reference manual.* – Oak Ridge National Laboratory, TN, USA, mai 1995.

[35] V. Getov, S. Flynn Hummel et S. Mintchev. – High-performance parallel programming in Java: Exploiting native libraries. *Concurrency: Practice and Experience*, vol. 10, n° 11-13, 1998, pp. 863–872.

[36] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta et J. L. Hennessy. – Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *In : Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pp. 15–26. – mai 1990.

[37] A. Gontmakher et A. Schuster. – Java consistency: Non-operational characterizatons for Java memory behavior. *ACM Transactions on Computer Systems*, vol. 18, n° 4, novembre 2000, pp. 333–386.

[38] J. Gosling, W. Joy et G. Steele Jr. – *The Java Language Specification*. – Reading, Massachusetts, Addison-Wesley, 1996.

[39] E. Hagersten, A. Landin et S. Haridi. – DDM-a cache-only memory architecture. *Computer*, vol. 25, n° 9, 1992, pp. 44–54.

[40] H. Hellwagner et A. Reinefeld (édité par). – *SCI: Scalable Coherent Interface. Architecture and Software for High-Performance Compute Clusters*. – Springer-Verlag, 1999, *Lect. Notes in Comp. Science*, volume 1734.

[41] High Performance Fortran Forum. – *High Performance Fortran language specification*. – Rice University, Houston, Texas, octobre 1996. Version 2.0.

[42] IEEE. – *Standard for Scalable Coherent Interface (SCI)*, août 1993. Standard no. 1596.

[43] IEEE Standards Department. – *1003.4d8 POSIX System application program interface : Threads extensions [C language]*, 1994.

[44] L. Iftode et J. P. Singh. – Shared virtual memory: Progress and challenges. *Proceedings of the IEEE*, vol. 87, n° 3, mars 1999.

[45] L. Iftode, J. Pal Singh et K. Li. – Scope consistency: a bridge between release consistency ad entry consistency. *In : Proc. 8th Ann. Symp. Parallel Algorithms and Architecture*, pp. 277–287. – 1996.

[46] Intel. – Streaming SIMD extensions driver home page. – Disponible à l'URL `http://developer.intel.com/vtune/optidrvr`.

[47] A. Itzkovitz et A. Schuster. – MultiView and MilliPage – fine-grain sharing in page-based DSMs. *In : Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 215–228. – février 1999.

[48] A. Itzkovitz, A. Schuster et L. Shalev. – Thread migration and its application in distributed shared memory systems. *J. Systems and Software*, vol. 42, n° 1, juillet 1998, pp. 71–87.

[49] Y. Jégou. – The migrating tasks: An execution model for irregular codes. *In : Proc. of Euro-Par'96 Parallel Processing*, éd. par L. Bougé, P. Fraigniaud, A. Mignotte et Y. Robert, pp. 562–570. – Lyon, France, août 1996.

[50] Y. Jégou. – Controlling distributed shared memory consistency from high level programming languages. *In : Proc. 5th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '00)*. Held in conjunction with IPDPS 2000. IEEE TCPP., pp. 293–300. – Cancun, Mexico, mai 2000.

[51] Y. Jégou. – *Loop-level parallelism and owner-compure rule on Mome, a relaxed consistency DSM*. – Research Report n° RR-4058, Rennes, France, IRISA/INRIA, décembre 2000.

[52] P. Keleher. – The relative importance of concurrent writers and weak consistency models. *In : 16th Intl. Conf. on Distributed Computing Systems*. – Hong Kong, mai 1998.

[53] P. Keleher, A. L.Cox, S. Dwarkadas et W. Zwaenepoel. – An evaluation of software based release consistent protocols. *J. Parallel and Distrib. Comp.*, vol. 26, n° 2, septembre 1995, pp. 126–141.

[54] D. Khandekar. – *Quarks: Distributed Shared Memory as a Basic Building Block for Complex Parallel and Distributed Systems*. – Master's thesis, University of Utah, mars 1996.

[55] L. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas et M. Scott. – Vm-based shared memory on low-latency remote-memory-access networks. *In : Proc. 24th Intl. Symp. on Computer Architecture (ISCA) 1997*, pp. 157–169. – Denver, Colorado, juin 1997.

[56] J. Kuskin. – The Stanford FLASH multiprocessor. *In : Proc. 21st Ann. Int'l Symp. Computer Architecture*. pp. 302–3013. – Los Alamitos, CA, 1994.

[57] L. Lamport. – How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, vol. C-28, n° 9, septembre 1979, pp. 690–691.

[58] P. Launay et J.-L. Pazat. – A framework for parallel programming in Java. *In : High-Performance Computing and Networking (HPCN '98)*. pp. 628–637. – Springer-Verlag, 1998.

[59] L. Lefevre. – *Mémoire Distribuée-Partagée sur Systèmes Parallèles et Distribués*. – Lyon, France, Thèse de doctorat, École Normale Supérieure de Lyon, 1997.

[60] D. Lenoski. – The Stanford Dash multiprocessor. *Computer*, vol. 25, n° 3, mars 1992, pp. 63–79.

[61] B. Lewis et D. J. Berg. – *Threads primer. A guide to multithreaded programming.* – Prentice Hall, 1996.

[62] K. Li. – *Shared Virtual Memory on loosely coupled multiprocessors*. – Thèse de doctorat, Yale University, 1986.

[63] K. Li. – IVY: a shared virtual memory system for parallel computing. *In : Proc. 1988 Int'l Parallel Processing*. pp. 94–101. – University Park, PA, 1988.

[64] K. Li et P. Hudak. – Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, vol. 7, n° 4, novembre 1989, pp. 321–359.

[65] R. Lottiaux et C. Morin. – Containers : A sound basis for a true single system image. *In : Proc. 1st IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGrid 2001)*, pp. 66–73. – Brisbane, Australia, mai 2001.

[66] S. Lucci, I. Gertner, A. Gupta et U. Hegde. – Reflective-memory multiprocessor. *In : Proc. 28th IEEE/ACM Hawaii Int'l Conf. System Sciences*. pp. 85–94. – Hawaii, janvier 1995.

[67] J. Maassen, T. Kielmann et H. Bal. – Efficient replicated method invocation in Java. *In : Proc. of the ACM 2000 Java Grande Conference*, pp. 88–96. – San Francisco, CA, juin 2000.

[68] C. Maples et L. Wittie. – Merlin: a superglue for multicomputer systems. *In : Proc. COMPCON'90*. pp. 73–81. – Los Alamitos, CA, 1990.

[69] E. Mascarenhas et V. Rego. – Ariadne : Architecture of a portable threads system supporting mobile processes. *Software : Practice & Experience*, vol. 26, n° 3, mars 1996, pp. 327–356.

[70] K. L. McMillan. – *Symbolic model checking*. – Kluwer Academic Publishers, 1993.

[71] D. Mentré. – *Une méthode de construction de mémoires partagées intégrant spécification, vérification et réalisation*. – Thèse de doctorat, Université de Rennes 1, 2000.

[72] Message Passing Interface Forum. – *MPI : A message-passing interface standard*, mars 1994. available from `http://www.netlib2.cs.utk.edu`.

[73] Motorola. – Motorola AltiVec technology : Home page. – Disponible à l'URL `http://www.mot.com/SPS/PowerPC/AltiVec`.

[74] F. Mueller. – A library implementation of POSIX threads under Unix. *In : Proceedings of the USENIX Conference*, pp. 29–41. – 1993.

[75] F. Mueller. – Distributed shared-memory threads: DSM-Threads. *In : Proc. Workshop on Run-Time Systems for Parallel Programming (RTSPP)*, pp. 31–40. – Geneva, Switzerland, avril 1997.

[76] F. Mueller. – On the design and implementation of DSM-Threads. *In : Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pp. 315–324. – juin 1997.

[77] F. Mueller. – Adaptive DSM-runtime behavior via speculative data distribution. *In : Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*. pp. 553–567. – San Juan, Puerto Rico, avril 1999.

[78] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse et J. M. van Staveren. – Amoeba – a distributed operating system for the 1990s. *IEEE Computer*, vol. 23, mai 1990, pp. 44–53.

[79] G. Muller et U. P. Schultz. – Harissa: A hybrid approach to Java execution. *IEEE Software*, vol. 16, n° 2, 1999, pp. 44–51.

[80] R. Namyst. – *PM2: an environment for a portable design and an efficient execution of irregular parallel applications*. – Thèse de doctorat, Univ. Lille 1, France, janvier 1997.

[81] R. Namyst et J.-F. Méhaut. – PM2 : Parallel Multithreaded Machine; A computing environment for distributed architectures. *In : Parallel Computing (ParCo '95)*. pp. 279–285. – Elsevier Science Publishers, septembre 1995.

[82] N. Nitzann et A. Schuster. – Transparent adaptation of sharing granularity in multiview-based dsm systems. *In : Proc. 15th Intl. Parallel and Distributed Processing Symposium (IPDPS 2001)*. – San Francisco, CA, avril 2001. Extended proceedings in electronic form only.

[83] B. Nitzberg et V. Lo. – Distributed shared memory: A survey of issues and algorithms. *IEEE computer*, vol. 24, n° 8, septembre 1991, pp. 52–60.

[84] M. Philippsen et M. Zenger. – JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, vol. 9, n° 11, novembre 1997, pp. 1125–1242.

[85] F. Pong et M. Dubois. – Verifications techniques for cache coherence protocols. *ACM Computing Survey*, vol. 29, n° 1, mars 1997, pp. 82–126.

[86] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham et S. Watterson. – Toba: Java for applications – a way ahead of time (WAT) compiler. *In : Proc. of the Third Conference on Object-Oriented Technologies and Systems*, pp. 41–53. – Berkeley, CA, juin 1997.

[87] J. Protic, M. Tomasevic et V. Milutinovic. – Distributed shared memory: concepts and systems. *IEEE Paralel and Distributed Technology*, 1996, pp. 63–79.

[88] L. Prylli et B. Tourancheau. – BIP : A new protocol designed for high performance networking on Myrinet. *In : 1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*. Held in conjunction with IPPS/SPDP 1998. IEEE, pp. 472–485. – Springer, avril 1998.

[89] U. Ramachandran et M. Y. A. Khalidi. – An implementation of distributed shared memory. *Software practice and experience*, vol. 21, nº 5, mai 1991, pp. 443–464.

[90] O. Reymann. – *Support d'exécution parallèle fondé sur des mécanismes de mémoire distribuée virtuellement partagée*. – Lyon, France, Thèse de doctorat, École Normale Supérieure de Lyon, 1999.

[91] J.-M. Rifflet. – *La communication sous Unix. Applications réparties.* – Ediscience International, 1994.

[92] R. Samanta, A. Bilas, L. Iftode et J. P. Singh. – Home-based SVM protocols for SMP clusters: design and performance. *In : Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pp. 113–124. – février 1998.

[93] D. J. Scales et K. Gharachorloo. – Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. *In : Proc. 7th Intl. Conf. on Architectural support for programming languages and operating systems*, pp. 174–185. – Cambridge, MA, octobre 1996.

[94] I. Schoinas. – Fine-grain access control for distributed shared memory. *In : Proc. 6th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*. pp. 297–306. – New York, NY, 1994.

[95] US National Science et Technology Council. – *High performance computing and communications : Advancing the frontiers of information technology*. – Rapport nº DOE/ER/25232–T1-Pt.1, Washington, DC (United States), US Departement Of Energy, Office of Energy Research, décembre 1997. Serial Number 98002025885.

[96] US National Science et Technology Council. – *Computing, information, and communications : Technologies for the 21. Century.* – Rapport nº DOE/ER/25232–T1-Pt.2, Washington, DC (United States), US Departement Of Energy, Office of Energy Research, décembre 1998. Serial Number 98002025886.

[97] M. Snir, S. Otto, S. Huss-Lederman, D. Walker et J. Dongarra. – *MPI : The complete reference*. – MIT Press, 1995.

[98] E. Speight et J.K. Bennett. – Brazos: A third generation DSM system. *In : Proc. of the USENIX Windows/NT Workshop*, pp. 95–106. – août 1997.

[99] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy et M. Scot. – CASHMERE-2L: Software coherent shared memory on a clustered remote-write network. *In : Proc. 16th ACM Symposium on Operating System Principles (SOSP '97)*, pp. 170–183. – Saint-Malo, France, octobre 1997.

[100] W. Shi W. Hu et Z. Tang. – JIAJIA: An SVM system based on a new cache coherence protocol. *In : Proc. High Performance Computing and Networking (HPCN'99)*, pp. 463–472. – Amsterdam, Netherlands, avril 1999.

[101] A. Wilson, R. LaRowe et M. Teller. – Hardware assist for distributed shared memory. *In : Proc. 13th Int'l Conf. Distributed Computing Systems*. pp. 246–255. – Los Alamitos, CA, 1993.

[102] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh et A. Gupta. – The SPLASH-2 programs: Characterization and methodological considerations. *In : Proc. 22nd Annual Int'l Symp. on Comp. Arch.*, pp. 24–36. – Santa Margherita Ligure, Italy, juin 1995.

[103] W. Yu et A. Cox. – Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, vol. 9, n° 11, 1997, pp. 1213–1224.

[104] S. Zhou, M. Stumm et T. McInerney. – Extending distributed shared memory to heterogeneous environments. *In : Proc. 10th Int'l Conf. Distributed Computing Systems*. pp. 30–37. – Los Alamitos, CA, 1990.

[105] Y. Zhou, L. Iftode et K. Li. – Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory. *In : Proc. 2nd Symp. Operating Systems Design and Implementation*, pp. 75–88. – octobre 1996.